Integer-time burst-level simulation techniques for very high speed communication networks

Stephen D. Cusack

Doctor of Philosophy
University of Edinburgh
2000

For my mother

Margaret Mary Cusack

1942-1996

Abstract

This thesis presents a collection of novel techniques that improve the experimental runtimes of high speed communication network simulations.

Broadband networks, and the applications made possible by these, provide a challenge to the traditional techniques of computer network simulation. Technologies such as Asynchronous Transfer Mode (ATM) offer high bandwidth connections with guaranteed Quality of Service (QoS) parameters for network users. Using simulation to model such networks is a computationally expensive process, especially to ensure that statistically valid results are produced.

This thesis presents a technique which abstracts the level of detail of a cell-based network to the *burst level*, where a burst describes a group of cells in transmission. The use of the burst level is designed to reduce the number of events which must be processed to perform a network simulation. To leverage the power of today's inexpensive, high performance microprocessors, the techniques presented exclusively use all-integer arithmetic. The use of integer arithmetic provides an inherent performance gain over floating point arithmetic. The use of 64-bit integer arithmetic is very desirable, which is a completely realistic goal for the next generation of microprocessors.

Techniques using integer arithmetic to multiplex, queue, demultiplex and switch bursts of cells are presented. Each operation is presented as a simulation object integrated into an efficient C++-based object-oriented bespoke simulation environment. The accuracy and performance issues for each object are explored, in comparison with an efficient cell level simulator also developed in the work.

Detailed investigation of the proposed techniques highlights two core operations, which are then further optimised as integer techniques (by removing the integer divide operation). The revised integer techniques are shown to improve the performance of the simulation objects, while preserving the accuracy of the techniques.

Three basic experiments are presented in the thesis to show how non-trivial simulations can be constructed from the core simulation objects presented. The performance and accuracy implications of each experiment are analysed and used to provide guidance for further work based on the techniques presented.

Acknowledgements

Firstly, I would especially like to thank my supervisor, Gordon Brebner, for all of his support, motivation and all round good natured encouragement of me and my work. Gordon helped turn my work and ideas into the thesis as presented, which would not have happened without his friendly drive and determination. I would also like to thank my previous supervisor, Rob Pooley, for helping to inspire the work and lay the foundations.

It is impossible to completely convey the enormous gratitude I have for the support and friendship given to me by my family and friends both throughout this work and through some extremely difficult personal circumstances. It is not an overstatement to say that without all of the support I have received, this work would never have been completed. Special mention must go to my Dad and brothers who were always available whenever I needed their help, my office mate and good friend Graham Jones who helped keep me sane and my friends Stuart Hartley, Julian Cawston, Callum McLean, Nancy Steele and Frances Wignall whose friendship and advice have been absolutely invaluable. I will always be particularly indebted to Julian Cawston and Roseanne Anthony for their incredible generosity in putting a roof over my head as I completed this work.

I have been very fortunate to have made many friends during my time in Edinburgh who I have met through my work and my love of climbing the hills and mountains of Scotland. Many thanks to Anna, Iain, Paul, Liz, Helen, Rob, Al, Mick, Julie, Ian, Ingrid, Mike, Mu, Sanjay, Ian, Ed, Tim and Calum. It has been a lot of fun and I've "bagged" over 180 Munros!

This work was partially supported by an EPSRC research studentship and an EPSRC CASE award in collaboration with BT plc.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The continual improvement in computer technology, and the associated reduction in cost, has led to the increased use of information technology in everyday life. The explosion in the usage of the *Internet* as witnessed in the 1990s, and the associated unbelievable stock market valuations of "Dot Com" companies, indicate that this trend is likely to continue. In the near future, the Internet will be heavily used as a medium for transacting commerce, as well as for entertainment and education purposes. The transition of the personal computer from being an esoteric and unfriendly "box" to an "information appliance" has been driven by the widespread desire to "get on the net".

To be capable of handling the expected volume of traffic, broadband technologies capable of delivering gigabit rate communication speeds have been developed for use in the backbone of the Internet. Provision of high-speed reliable networks has opened up the range and diversity of applications which can communicate over the networks. Applications traditionally with high bandwidth requirements, such as video on demand, can be realised with such technology. The addition of negotiated Quality of Service (QoS) parameters for data in the network increases the reliability and quality of the networking service which can be offered to users. The potential rewards of providing high-bandwidth high-quality networks, and the feature-rich services which can use them, are immense. A great deal of research and development effort is focused on making such networks a reality.

Cell-based networking technologies, in particular, have shown a great deal of promise as being the likely technologies to support gigabit rate networking. A cell-based network transmits fixed length cells of data which are routed by cell switches within the network. A network user transmits Quality of Service requirements to the network, which then admits the user if the parameters can be accommodated. An example of a cell-based networking technology capable

of offering gigabit transmission speeds, as well as Quality of Service guarantees, is Asynchronous Transfer Mode (ATM). ATM has been widely adopted as a broadband networking solution and much research has gone into its design and implementation. Simulation and modelling have been used extensively as tools in this work.

Broadband networks, such as ATM, and the applications designed to exploit the bandwidth available, provide a challenge for the techniques of computer network simulation and modelling. Modelling the operation of the underlying network hardware, as well as the dynamic behaviour of the data carried, is extremely challenging. Mathematical analysis can be used to model such networks, but producing models of adequate detail which can be solved is very difficult. Computer simulation models of the networks can offer a far more detailed approach, but the compute times necessary to perform such simulation studies can be prohibitive.

Various techniques have been proposed as means of improving the performance of computer network simulation experiments. This thesis presents a collection of novel simulation techniques which have been designed to improve the runtimes for simulation of very high speed communication networks. The emphasis of the work is on providing techniques for the fast and efficient simulation modelling of cell-based networks such as ATM. Although the techniques have been designed with this type of network in mind, the techniques are shown to have broader applicability in computer network simulation. The techniques developed have been implemented within an efficient sequential object-oriented bespoke discrete event simulation environment.

Two fundamental approaches are used in the design of the techniques presented in this work. Firstly, the level of detail modelled in the technique is abstracted to the burst level. A burst represents a group of related cells and is the smallest quantity considered in the simulation. The advantage of the burst level is that it reduces the total number of events which must be processed in a simulation when compared to an equivalent cell-level simulation. Coupling the decreased event count with the efficient processing of bursts in the simulator is a means of improving simulator performance.

The second fundamental design decision is to use integer arithmetic throughout the simulator implementation. The rationale for this decision is that today's fast microprocessor designs offer an inherent performance advantage for integer arithmetic when compared to the floating point equivalents. The widespread use of superscalar architectures, with processor clock speeds for mainstream architectures reaching 1GHz, makes even the "ordinary" personal computer a very

powerful tool for simulation research. The next generation of microprocessor architectures will offer native 64-bit computation and even higher processing speeds. Designing the simulation techniques presented in this work to maximise integer performance makes them well placed to leverage the incredible power of these processors.

Integer techniques are presented for the fundamental operations necessary to perform burst-level simulation of very high speed networks in an efficient discrete event simulation environment. The techniques described are designed to be portable, in that they are independent of the precise simulation framework used. Alternative event list techniques, or even the use of parallel discrete event simulation techniques, are possible. The techniques presented are concerned with the fast and efficient transport of bursts between objects in a simulation model. Other methods which can improve simulation runtimes (such as statistical methods to bias the probability distributions used in the model) which are independent of the simulation framework used, are thus applicable to the techniques.

The structure of the thesis is as follows.

Chapter 2 overviews the techniques of simulation and modelling, with particular emphasis on the application of these techniques for communication network modelling. An introduction to Asynchronous Transfer Mode (ATM) cell-based networking is given, along with examples of simulation and modelling techniques which have been used in the research and development of such networks.

Chapter 3 presents the techniques developed in this work for integer-time burst-level simulation of communication networks. The chapter explores the issues raised when one wishes to use discrete event simulation, and describes the design of an efficient bespoke C++-based object-oriented simulation environment. The advantages of an object-oriented design methodology are stressed, along with other issues such as the choice of event list management which influence the performance of the simulator produced. The chapter defines the notion of a burst, as used in this work, alongside the exclusively integer techniques for provision of the fundamental operations of burst multiplexing, demultiplexing, queueing and switching.

Analysis of the techniques presented in Chapter 3 highlights two core operations whose accuracy and performance are crucial. Chapter 4 explores the accuracy and performance issues of *burst creation* and *burst splitting* which are the two core techniques of interest. Examination of the performance issues for each technique, coupled with an investigation of the cost of integer instructions in modern microprocessors, leads to the development of further optimised in-

teger versions of the core techniques. Crucially, the integer divide operation is shown to have punitive throughput and latency costs when considering runtime performance. The revised core techniques remove the integer divide operation where possible and are shown to give a performance benefit while not degrading accuracy.

Chapter 5 presents an in-depth analysis of the accuracy and runtime performance of each fundamental simulation object described in Chapter 3. Versions of each object using the original and revised core integer techniques are compared against the runtimes and results from an efficient cell-level simulator also developed in the work. The cell-level simulator shares the same efficient simulation environment used to develop the burst-level simulators used in this chapter.

The simulation objects presented in Chapter 3, revised in Chapter 4 and analysed in Chapter 5, are designed to provide the fundamental building blocks for efficient simulation models of high speed networks. Chapter 6 presents three simulation experiments, each showing the operation of the burst-level simulation objects in different scenarios. The first experiment is an example of a cell-based network requiring a model with many interconnected simulation objects. The second experiment examines the burst-level techniques when they are subjected to very highly multiplexed loads. The third experiment models the operation of a very high speed communication network used in a network of workstations. In particular, the third experiment demonstrates how the techniques may be used for simulation models other than cell-based networks.

Finally, Chapter 7 summarises the work performed in this thesis and outlines some interesting directions for future work based on the techniques presented.

Chapter 2

Background

2.1 Introduction

The aim of this chapter is to introduce simulation and modelling, particularly as employed in the field of computer communications. An overview of simulation is given first, with emphasis placed on the different techniques used to simulate communication systems. The simulation of very high speed communication networks, with particular emphasis on cell-based networking, is the central focus of this thesis. *Asynchronous Transfer Mode* (ATM) is an example of a high bandwidth cell-based networking technology, which has been the subject of much research effort. A brief introduction to ATM networking is presented, along with an overview of simulation techniques used to study the technology. Finally, a summary of the chapter is given.

2.2 Simulation and modelling

Computer simulation (as described by Fishwick[38]) is:

> "The discipline of designing a model of an actual or theoretical physical system, executing the model on a digital computer, and analysing the execution output."

The scope for simulating real-life systems is vast, and the use of simulation has many advantages. Simulation may be used to model an existing system in an attempt to gain insight into its behaviour. The real life system may function too rapidly, or be too dangerous or expensive to monitor, so simulation may be used to gain understanding of the factors influencing behaviour. On the other hand, simulation can be used in the design of new systems or processes in an effort to avoid costly mistakes. Simulation allows modellers to vary the parameters

of their proposed systems to see how they behave. Such a study can predict potential "bottlenecks" and help to diagnose any problems (such as incorrect behaviour or "unexpected" quirks). As the system only exists in the simulator (as a model) at the design phase, changes can be rapidly made to the design and the behaviour when problems are highlighted. A comprehensive and methodical simulation study can iteratively progress the design of a system in a cost-effective manner.

Simulation is a very powerful tool for examining the behaviour of a system, but the insight gained has to be carefully balanced against the assumptions built into a model, as well as the similitude between the model and the "real world" system. It is important to follow a simulation *methodology* (as described by Fishman[37] for example) when designing and performing a simulation study so that the results are meaningful.

The first step is to determine whether simulation is actually required or not. If the problem to be investigated for the chosen system can be solved analytically, then this is the best course of action. If an analytical approach cannot be used, the next question is to determine whether the problem is amenable to computer simulation techniques. If so, the system under consideration should be defined in terms of entities, attributes, class relationships among entities, dynamic relationships, input stimuli and performance criteria relevant to the problem. If the problem is not amenable to simulation then some other technique must be found to provide a solution.

The next step (if simulation is chosen for the problem) is to choose a simulation language and modelling approach. If the simulation modeller has expertise with a particular language and simulation system, the system can be modelled in a way which is consistent with the selected language. If no such expertise exists, a modelling approach and language must be selected before the system can be modelled. The choice of a bespoke simulation program for the study requires the additional provision of simulation primitives which must be checked for predictable and consistent behaviour to ensure accurate modelling of the system. Once a simulation language and approach have been chosen the model can be suitably programmed. Careful *verification* of the simulation model implementation is essential at this stage to ensure that the model, as programmed, accurately represents the system under investigation. Trapping programming and logical errors during development will help to ensure reliable results when the simulation runs are performed.

Once a verified simulation model of the system has been produced, suitable input data to drive the simulation runs must be selected. If real world data for the system under investigation is available this can be analysed to provide the necessary input stimuli for the model. The availability of real world data enables the behaviour of the model to be *validated* against the behaviour of the real world system. Greater confidence can be placed in the simulation model if it produces output data consistent with that observed in the real world system for the same input parameters. If no such real world data exists, the investigator must estimate the probability distributions and parameter values for each of the inputs in their simulation model.

The next step is to design the experiments to be used to investigate the chosen problem for the modelled system. The accuracy of the final results will be dependent on the length and number of experiments performed with the simulation model. To achieve a very high degree of accuracy, a great deal of compute time may be required which may be prohibitive. It is worth spending time before embarking on the simulation runs to ensure that the test statistics required, as well as the test procedure used, provide adequate data for the problem. Experiments may be *replicated* with greater instrumentation if the output data is inadequate, but this will lead to an increase in the time required. If Monte Carlo techniques, or pseudo-random number distributions, are used care must be taken to ensure that the random number generator used is free from known defects.

The final stage is to perform data analysis on the results of the simulation runs. Statistical techniques should be used to calculate the parameters of interest along with their variances and estimated confidence intervals. Depending on the analysis of the results, further experimental runs, or even entirely new simulation experiments, may need to be performed to ensure confidence in the results.

2.3 Simulation techniques for communication networks

2.3.1 Introduction

Computer communications technology has rapidly evolved with advances in semiconductor design and the development of transmission technologies such as fibre optic cables. The art of computer simulation has also kept pace and evolved with the changes in technology, harnessing the power of modern computer systems. Technological progress has led to cheaper and increasingly powerful microprocessors which can access large amounts of relatively cheap physical memory. Today's

"ordinary" desktop PC is as capable as the supercomputers and workstations[46] of even a decade ago. Demand for the *Internet* has helped drive the development of cost effective high bandwidth networking technologies[53]. Other advances, in the field of parallel computing for example, have also increased the capability for performing detailed simulation studies.

Ever since the first days of telephone and data networking[93], simulation and modelling have been used as a tool to help design and dimension networks capable of carrying the traffic of the day. As networking technology has progressed, the variety of data traffic the network is capable of carrying has increased. Applications which would have been impossible a few years ago, such as *collaborative virtual environments*[49] and *video on demand*[66], are becoming a reality due to high speed reliable communication networks which offer guaranteed *Quality of Service* (QoS). The new traffic types, and the fast underlying networks, provide fresh challenges for today's network planners. The old modelling theories and techniques (such as the 1909 Erlang B and C models[35] for predicting blocking and delay in the network) are no longer applicable to modern networks and the applications enabled by the bandwidths available (the "paradigm shift" to today's networks as described by Wirth[113]). Simulation and modelling are now increasingly important for helping to design and dimension teletraffic networks capable of handling this traffic.

Simulation is widely used in both industrial and academic research on telecommunications networks. Typically, simulation is the tool used to validate and assess the correctness of analytical techniques (eg. buffer utilisation in cell-based networks[106]) or to quantify the likely performance of expected network traffic types and protocols (eg. video-on-demand carried by the ATM *Available Bit Rate* (ABR) service[102]). Commercial tools are widely used for this work (eg. OpNet for ATM[116]) due to convenient libraries of traffic sources (eg. *Variable Bit Rate* (VBR) traffic) and network protocols (eg. TCP/IP and the *ATM Adaption Layers* (AAL)) being available.

For a simulation environment to be useful for communications research, a requirement is that *closed-loop* operation must be supported (ie. network nodes can respond to feedback from the network) as well as *open-loop* capability for producing end-to-end statistics (eg. assessing a traffic aggregation model[55]). As described by Nichols[85], open-loop models are easy to produce and simulate, but can be shown to miss the detailed structure of the system under investigation.

The following sections summarise the most widely used approaches for the modelling and simulation of communication networks.

2.3.2 Mathematical analysis

Analysis can be used to produce mathematical models of communication systems, or indeed parts of them. Queueing theory and probability theory are used to construct the models, which are then solved to produce the results for the parameters being measured. The difficulty of this technique is producing models which *can* be solved and which model the system under investigation in sufficient detail. King[68] provides an excellent introductory text on the application of queueing theory and probability theory to communication system modelling.

The use of probability theory allows for a system to be described by a few parameters, rather than a detailed set of input data. Network models are constructed as *queueing networks* based on queueing theory. As described by King[68], a queueing system can be specified in terms of six component parts.

1. The arrival process: which is a stochastic process describing how jobs arrive in the system from the outside world.

2. The service process: also a stochastic process which describes the length of time that a server is occupied by a job.

3. The number of servers and their rates of service.

4. A queueing discipline: effectively rules for deciding which job or jobs to serve.

5. A waiting room for jobs awaiting service.

6. The customer population (ie. the population describing the jobs which may enter the system).

Queueing systems are typically described by *Kendall's notation* which is of the form $A/S/c/n/p$ where:

A is the arrival process.

S is the service process.

c is the number of servers.

n is the size of the waiting room (assumed to be ∞ if omitted).

p is the population of potential customers (assumed to be ∞ if omitted).

The simplest queueing system is the $M/M/1$ system where a single server gives exponentially distributed service times to customers that arrive in a Poisson stream. The M notation means that the process is *Markovian*. A Markov process is a stochastic process in which the distribution at any time in the future depends only on the current state of the process, and not on how the state was reached. The simplicity of Markovian processes when calculating state probabilities is very useful in queueing theory.

The goal of a queueing network model is to produce a *closed-form* expression which can be solved exactly. If this is not possible, then numerical techniques can be used to calculate the performance measures of interest in the system.

2.3.3 Discrete Event Simulation (DES)

Discrete event simulation works by decomposing systems into a collection of *objects* which may interact in certain defined ways. Each such interaction is an *event*. Events may only happen at a certain time, so the the progression of simulated time is modelled in discrete steps, rather than as a continuum. Two common approaches to discrete event simulation are *event-based* modelling and *process-based* modelling. DES is central to the work of this thesis, and various facets will be covered in depth in later chapters. In particular, the application of DES to communication network modelling is covered in more detail in Chapter 3 (with process and event-based modelling detailed in Section 3.2.2.2). With this in mind, the techniques are just introduced in summary form in this section.

Event-based modelling essentially decomposes a system into a collection of events, each of which alters the state of the entire system. A simulation run consists of processing the event list which is a *calendar* of events waiting to happen. Events are added to the list in order of increasing simulation time. New events are scheduled, and appropriately added to the list, as events already in the list are processed.

Process-based modelling decomposes a system into a number of *processes*, each of which represents the *activity* of some part of the system under consideration. Each process may compete for shared resources or communicate with other processes to progress the state of the simulation. Discrete events scheduled in a time-ordered global event list are used to progress the state of each process. The event at the head of the global event list represents the current simulation time. The current event is processed by removing it from the global event list and advancing the state of the process for which the event was scheduled.

Discrete event simulation (both event and process-based) is well covered in the

literature, with good introductions given by Mitrani[83], Franta[40], Bratley[16], MacDougall[77] and Schriber[95].

2.3.4 Parallel Discrete Event Simulation (PDES)

Parallel discrete event simulation[42] (PDES) is one means of improving the run-time performance of discrete event simulations. The model being simulated is broken up into separate sub-models, each of which can be simulated as a sequential simulation which can interact with other parts of the model. Each sub-model is called a *Logical Process* (LP) and a PDES program is a collection of interacting LPs. An LP communicates with another LP by transmitting a message that contains an event. A timestamp in the message indicates the simulation time at which the event is to be executed on the destination LP. When a message arrives at an LP (which may be scheduled by another LP or by the LP itself), the event is scheduled for execution. This normally involves inserting the event into some time-ordered event list local to the LP. Each LP maintains a *Local Virtual Time* (LVT) which is its own simulation clock, the value of which is the timestamp of the the last event executed. A *Global Virtual Time* (GVT) simulation clock is also maintained, whose value is the minimum of all the LVTs and the timestamps of messages in transit in the simulator. The progress of the simulation is recorded by the GVT.

Each LP must process events in order of increasing timestamp to maintain the temporal relationships of the simulation model. Ensuring that this is the case is the fundamental difficulty of parallel discrete event simulation. Two methods commonly used to synchronise the actions of each LP can be outlined as follows.

The first common synchronisation method used in PDES is the *conservative*[22] simulation protocol. A conservative PDES prevents out of time order execution of events at each LP. This is achieved by each LP marking events as either *safe* or *unsafe*. A safe event can be executed, as it is guaranteed that no event with a smaller timestamp will subsequently arrive. An unsafe event has no such guarantee, and so is blocked until it can be marked as safe. An event can be marked safe if an event received previously is marked safe (usually when an event is received from another LP), or when a new event is received and guaranteed to be safe.

To make this protocol work, messages are sent from one LP to another in increasing timestamp order. The receipt of a message from an LP is a guarantee that no messages with smaller timestamps will be be received afterwards from that LP. If an LP has received messages from all other LPs, such that each had

a timestamp greater than an event marked as unsafe, the event can be marked as safe. Deadlocks are prevented in the system by each LP periodically sending *null* messages, simply containing a timestamp and no event. The null events are used to update the minimum timestamp of messages from a particular LP.

The second most common synchronisation method used in PDES is the *optimistic*[64] simulation protocol (often referred to as *TimeWarp*). In an optimistic PDES, all events are regarded as safe and executed accordingly. If a message arrives with a timestamp smaller than the LVT (ie. an out of time order message), events which have been executed already must be *rolled back* until the LVT is smaller than that of the out of time order message.

The rollback process may involve the sending of *anti-messages* to cancel messages which have been sent to other LPs in error (due to the incorrect processing of events). When an anti-message arrives at an LP, the event queue is checked to see if the erroneous event has been executed yet. If it has not been executed, the event is deleted from the queue and no other action needs to be taken. If the event has been executed, the state of the LP must be rolled back to a time earlier than the timestamp of the incorrectly processed event. Either *lazy* or *aggressive* strategies[74] may be used for message cancellation. Lazy cancellation waits until a previously scheduled message is deemed to be erroneous before sending a suitable anti-message. This may lead to an increased number of state rollbacks due to the increased likelihood of erroneous messages being processed. Aggressive cancellation immediately cancels every possibly erroneous message scheduled by an LP before it was rolled back. Messages which may later prove to have been correct are cancelled as well as erroneous ones.

Considering the complexity of the synchronisation techniques used in PDES, much research has been done to maximise PDES performance. Some examples of the research activities in this area are outlined below.

Cleary and Tsai[23] present a conservative algorithm which outperforms a TimeWarp simulator when many messages are scheduled in a simulation (as is the case for high speed communication networks). The simulator dispenses with null messages, and instead randomly selects an LP which consumes all its waiting messages on each *link* connecting it to other LPs. The events are then executed and the LP suspends when an input link contains no more messages. The LVT of the process is then set to the lowest message time on its input links.

Das[32] presents a cost model of optimistic execution in TimeWarp which can be used to predict the size of a suitable time window for limiting the degree of optimism at each LP. Each LP may only process events in the time window

spanning the GVT to the GVT plus the window size.

Bisset[13] presents an adaptive synchronisation protocol for optimistic PDES which uses a neural network to determine if an event can be marked as safe. Adapting to changes in the behaviour of the model was found to improve performance by the reduction of state rollbacks at each LP.

Steinman[98, 99] describes the concept of the *event horizon* which can be used to prevent the sending of anti-messages in an optimistic PDES, and thus control optimism. Each LP may process events up to the event horizon without fear of out of time order messages arriving. The event horizon is determined by each LP optimistically processing its own local events, but deferring the sending of messages until it has reached a simulation time where the next event to process will be an unsent message. The time reached is the *local event horizon.* Once each LP has determined the local event horizon, a global minimum is determined and this becomes the global event horizon. Each node then rolls back to the global event horizon time (which is easy as no messages have been scheduled), and then sends the messages which are valid. Each LP can then commit to processing events up to this point.

Although the state of the art has progressed in PDES (with the technology argued to be mature enough for widespread use by Bhatt[10] and Lin and Fishwick[73]), producing efficient simulation models suitable for PDES is a difficult task. Bagrodia[7] outlines the steps required to produce a good PDES simulation model, as well as some of the pitfalls commonly encountered with the technique. Some researchers have looked at providing parallel simulation environments which attempt to hide the underlying complex parallel simulation synchronisation and parallel programming details from the user. Such an environment, *SPaDES* (Structured Parallel Discrete-Event Simulation), which uses TimeWarp, is presented by Teo and Tay[103].

A great deal of PDES research effort has assumed that shared-memory multiprocessor computers will be used for executing the simulation. The use of such expensive hardware benefits from the low latency, high bandwidth inter-process communication possible in these machines. As an alternative, the increasing power of desktop computers has lead to the investigation of PDES over *Network Of Workstation* (NOW) clusters[96]. The argument is that a NOW cluster is very cost effective, when compared to a shared-memory multiprocessor, due to the use of inexpensive personal computers and legacy local area network infrastructures (such as *Ethernet*). Scaling a NOW cluster is also very easy, compared to partitioning (or expanding) the capacity of a shared-memory multiprocessor.

The main difficulty with PDES on a NOW cluster is the high latency of the legacy interconnection networks used. Two means of limiting this drawback are either to reduce the number of messages sent in the simulation[67, 89] or to use parallel and independent local area networks in the NOW cluster[56] to improve the communication performance while keeping the cost minimal.

The use of PDES does not feature in the work presented in this thesis, but the potential application of the technique is discussed as future work in Chapter 7.

2.4 ATM Networking

2.4.1 An introduction to ATM networking

Asynchronous Transfer Mode (ATM) networking is a technology developed by the ITU-T (*International Telecommunications Union - Telecommunications*, formerly the CCITT) as part of the *Broadband Integrated Services Digital Network* (B-ISDN) for mixed telephone and data networks. Data rates of 155Mbs^{-1} and 622Mbs^{-1} are supported for B-ISDN implementation, with gigabit rates being supported in the future. ATM is an example of a *connection-oriented cell-based* network, where fixed length cells of information are sent between peers once a connection has been established between them.

The choice of the cell (which is a fixed length packet of bytes) as the smallest data unit transmitted has the following advantages.

- Multiplexing benefits. Traditional packet-based networks can suffer from *serialisation*, where shorter packets can get delayed behind longer packets on a shared line. A cell-based network multiplexes more fairly as the cells from each packet are intermingled by the multiplexing process.

- Consolidation of networks. Telephony and data traffic can be carried on the same cell-based networks as different *classes* of traffic.

- Fast switching capability. The fixed size of cells allows for the design of fast hardware to switch cells in the routing network.

- Multicasting capability. The complexity of multicasting to a number of receivers is reduced with cell-based networks.

In an ATM network, each cell is 53 bytes long, with a 5 byte header and a 48 byte data payload. The 48 byte length was chosen as a compromise between a data communications lobby that wanted 128 byte cells (to make ATM well suited for data traffic), and a telephony lobby that wanted a cell length of 16

bytes (which is optimal for voice samples in a telephone network). Transmission of user data packets in a cell-based network requires that each packet be broken into cell-sized chunks before transmission over the network. At the receiving end, the data from the cells must be reassembled into the original data packet before being passed to the user. Cells are routed through an ATM network by ATM *switches* which select the route of each cell based on information carried in the cell header.

Two different ATM cell header formats are supported. The first is used between a computer and the ATM network and is defined by the *User-Network Interface* (UNI) protocol. The second is used between switches inside the ATM network and is defined by the *Network Node Interface* (NNI) protocol. Each cell header contains a *Virtual Path Identifier* (VPI) and a *Virtual Channel Identifier* (VCI). The idea is that the network supplies *virtual paths* between switches, each of which is shared by a number of *virtual channels*. Any messages travelling on any virtual channel in a particular virtual path follow the same route. This enables intermediate switching nodes in the network to determine where to route each cell based only on the information in the VPI. The VPI is longer in an NNI cell than a UNI cell as it is assumed that there are more paths between switches than there are between one computer and the network.

When a computer establishes a communication session with the ATM network (ie. the *call setup* phase using the UNI protocols), it is assigned a VPI/VCI pair which is used to route the cells to their destination by the network. Negotiation of an agreed *Quality of Service* (QoS) for a communication session is covered in the UNI protocols, with the NNI protocols used to establish a route guaranteeing the requested quality of service within the network switches. Several classes of traffic types have been suggested for ATM networks which are summarised as follows.

- **Constant Bit Rate** (CBR). Traffic of this type is classed as *real-time*, with a constant data transmission rate (eg. a telephone call). The call setup procedure negotiates guaranteed bounds for transmission bandwidth, transmission data loss (ie. cell loss within the network) and delays experienced by CBR traffic.

- **Variable Bit Rate** (VBR), both real-time (rt-VBR) and non-real-time (nrt-VBR). VBR traffic varies the transmission rate during a communication session. Real-time VBR traffic (eg. video or audio) usually has tight constraints on delay and loss, whereas non-real-time VBR traffic (eg. prioritised multimedia electronic mail) has more flexible delay constraints. Both

types of VBR traffic may be *bursty*, which means that the transmission rate may fluctuate between the low and high limits allowed for the traffic for intervals of varying lengths. Depending on the VBR traffic type, the QoS guaranteed during call setup will give bounds for the bandwidth, cell loss and delay for the VBR traffic.

- **Available Bit Rate** (ABR). ABR traffic is intended to use spare bandwidth in communication channels. Guarantees are given for any cell loss encountered and a minimum bandwidth, but not for any delays. Flow control between the network and the communicating node is used to regulate the transmission of cells into the network depending on the network status.

- **Unspecified Bit Rate** (UBR). UBR traffic is given no guarantees for bandwidth, cell loss or delay. UBR traffic is typically a prime candidate for cell loss in congested buffers in the network.

Sending data over an ATM network requires the use of an *ATM Adaption Layer* (AAL) protocol rather than directly accessing the raw cell switching service. The AAL performs the *segmentation* of messages into ATM cells when sending and the *reassembly* of cells back into messages when receiving. Several AALs have been defined, with each specified for a different traffic type. The AALs currently defined are summarised as follows.

- **AAL1**: is designed to carry CBR traffic with strict delay bounds. Data is simply broken down into cells and sent on to the receiver. No error correction is implemented (as no delay can be introduced) but a sequence number allows for missing messages to be detected by the receiver. Buffering at the receiver is used to ensure that the time characteristics of the received messages are the same as those of the source messages.

- **AAL2**: is a protocol still under development which was meant to carry the VBR service. Stronger error correction, when compared with AAL1, was the main feature of AAL2. AAL2 has recently reemerged as an AAL for voice traffic.

- **AAL3/4**: was designed to carry the ABR and UBR services (ie. supporting connection-oriented and connectionless information transfer). The protocol handles the connection management and transport of the data allowing, for example, delivery of messages to be guaranteed or not. AAL3/4 allows several different channels to be multiplexed onto the same connection to the network.

- **AAL5**: is a simplified layer which has proved to be the most popular AAL for data transport and has largely superseded AAL3/4. AAL5 allows message delivery to be guaranteed or best-effort, with error detection applied to each message. Messages with errors may either be dropped or delivered with a warning. AAL5 assumes that the connection management and the multiplexing of data are handled by a higher level protocol, and thus more data may be carried in each ATM cell due to the reduction in the overhead required. AAL5 is widely used for the transport of TCP/IP traffic over ATM networks.

One of the anticipated benefits of the use of ATM is that each connection may be *statistically multiplexed.* Each QoS contract with a source has a peak cell transmission rate which must not be exceeded. Absolute QoS guarantees are assured if the peak rate of each connection is used as the basis for admitting connections. However, this wastes bandwidth as a statistical multiplexing gain is not exploited. If several VBR sources are using a connection, the total data rate at any one time may not match the bandwidth of the link. Exploiting this fact allows extra connections to be admitted to the link that would otherwise be excluded if *peak rate allocation* of bandwidth to each active connection was used. Services such as ABR and UBR (with TCP/IP typically used as the higher level protocol) are primarily designed to use the "wasted" bandwidth, while not infringing the QoS guarantees for VBR and CBR connections also using the network.

As ATM is an important networking technology it is well covered in both commercial and research literature. Good introductions to ATM and broadband communications in general are given by Partridge[88], De Prycker[33] and Brebner[17]. Händel *et al.*[54] give a more technical presentation of the ATM protocols. Specifications of the ATM protocols (such as UNI and NNI) are available from the ITU-T[63] as well as the *ATM Forum*[39] which is a consortium of ATM networking equipment manufacturers.

2.4.2 Parameters which influence the Quality of Service in an ATM network

As described in the previous section, a new call to an ATM network negotiates a set of Quality of Service parameters for its connection to the network. The parameters negotiated usually include bounds on cell loss and cell delay variation. Cell loss is typically specified as a *cell loss ratio* (CLR) which is the ratio of lost cells to transmitted cells in the network. Cell delay variation is a measure of how the inter-cell arrival time of a stream of cells is altered by the network

carrying those cells. The inter-cell arrival time can increase, for example, when the cell stream is contending for a busy communications link. A decrease in the inter-cell arrival time can be caused when cells from the stream are buffered and subsequently retransmitted somewhere in the network.

Depending on the higher level protocols used, both cell loss and cell delay variation will have an influence on the performance of the user application producing and receiving the cell traffic to and from the network. Cell loss may require retransmission of data packets from a communicating peer. Such retransmission may cause the application large delays in a high bandwidth network where many cells will be in transit on a link at any one time, especially when the cell loss renders data carried by subsequent cells incorrect. The influence of cell delay variation depends on the protocols and techniques used by the user application. Use of playback buffers and other such techniques can increase the robustness of the application to network induced cell delay variation. If such techniques can not be used, and the application has very tight constraints on the permitted cell delays, the effect of cell delay variation on the Quality of Service received by the user application can be severe. A real time audio visual data stream, for example, would have to discard video frames or distort the audio track if the delays in the incoming cell stream were severe. Of vital importance to ATM network service providers will be to ensure that QoS guarantees negotiated and paid for by their customers are satisfactorily upheld.

2.5 Modelling ATM cell-based networks

2.5.1 Introduction

Although the cell switching concept behind ATM networking is relatively straightforward, the protocols necessary for its operation, as well as the characteristics of the data traffic likely to be carried, have made ATM networks the focus of intense study. Much research has gone into attempts to mathematically model both the likely data traffic as well as the underlying network technology. The mathematical analysis of ATM networks has proven to be extremely difficult, and so a greater emphasis has been placed on simulation as a means of predicting the performance of such networks.

Simulating ATM networks is also a complicated proposition due to the cell-based nature of the actual hardware. Simulating the transmission of each cell in even a moderately sized network model can be very computationally expensive. Actual ATM networks are expected to have tiny cell loss ratios of the order of

10^{-12} or less. Using *Monte Carlo* methods for simulating such networks to give statistically valid results requires many repetitions of lengthy simulation runs.

Naturally, a great deal of research has concentrated on looking at ways and means of reducing the computational cost of simulating cell-based networks while maintaining good accuracy of the results obtained. Attempts to improve the efficiency of ATM simulations can generally be placed into three categories which relate to the level of detail simulated by each method. The three categories can be summarised as follows.

- **The cell level** is the most detailed form of cell-based network simulation. As would occur in a real network, the transmission of each cell is modelled in the simulation. As would be expected, this approach yields the most accurate results (as it closely follows the operation of real hardware) but suffers from the computational expense of simulation at this level of detail. The cell level is typically used to model complex protocol interactions in networks (where explicit cell delays and losses are critical) and the behaviour of cell-based networking hardware such as cell switches.

- **The burst level** is an abstraction of the behaviour at the cell level. Rather than consider each cell individually, the burst level groups related cells together into a *burst*, which is the smallest transmission unit considered in the simulation. Grouping cells together reduces the total number of events which must be simulated in a simulation run, with the aim of reducing the computational cost of each simulation performed. The burst level is typically used to model networks when the cell level is too detailed and the call level is too abstract for the simulation study. The burst level is a convenient abstraction to use for operations such as traffic policing and congestion control (eg. [21, 45, 72, 105, 112]) as it reduces the work necessary in the networking hardware. Modelling at the burst level is one means of assessing the various techniques.

- **The call level** is the level of detail which encompasses the broadest time-scale. A *call* to the network from a user application lasts for the duration of the communication session (which is the time of the connection to the network). The call level is thus an abstraction of the burst level, and allows for large periods of simulation time to be covered in reasonable compute times. The fine detail of the network, as well as the traffic carried, is not modelled in depth at this level. The call level is typically used to model the *operation and management* features of a real network. The effects

of different traffic policing policies, as well as resource allocation to the network, can then be assessed over long timescales.

The following sections describe various techniques which have been explored for ATM network simulation at each level of detail described. The final section describes techniques for improving simulator performance which can be used at any level of detail.

2.5.2 The cell level

The *CLASS*[1] (ConnectionLess ATM Services Simulator) simulation tool is an example of a time-driven, slotted, synchronous simulator (written in the C language) for modelling ATM networks. The rationale behind CLASS is that most objects within cell-level ATM network simulations are involved in some operation at any time instant. Rather than use a discrete event simulation methodology, simulation time is progressed in "slots", with the duration of one slot being set equal to the transmission time of one cell on the fastest link modelled. Other data rates are simulated as integer multiples of this time. At each time step, objects which have cells to send or receive are activated, and the transfers are performed. To avoid wasting time checking every simulated link at each time step, a *minimum scheduling sequence* is calculated based on the transmission speeds of each link. This ensures that at each time step, only the nodes which can be active at that time are checked for cell transfer activity. The approach used in CLASS gives good performance for networks simulating heavy loads. Some example applications of CLASS have been the study of fair queueing in ATM networks[2] and *Call Admission Control* (CAC) for different QoS classes in ATM networks[79].

YATS[108] is another example of a slotted ATM cell-level simulator, written using the C++ programming language. The slotted nature is a drawback of the system (as different line speeds are simulated as multiples of the basic time used in the simulation), but good performance is seen for basic simulation models. A wide variety of cell sources and various traffic management options are provided.

The *TeleSim* project[107] is a cell-level simulation tool implemented for both sequential and parallel[44] processing. Much work has focussed on the design of detailed cell-level traffic sources and workload models (eg. for self-similar video streams[9]) and on research into the parallel simulation techniques used. Recent work has looked at a new conservative algorithm (*Critical Channel Traversing* (CCT)[115]) which has shown improved performance for large ATM network simulations on shared-memory multi-processor computers. Simulation models are written in the *SimKit*[47] language which has been implemented in both C++

and Java. The C++ version allows the use of the parallel simulation kernels, whereas the Java version is restricted to the sequential version of the simulator only.

The design, performance and analysis of cell switches has been an ongoing research theme in the development of ATM networks. Simulation at the cell level is crucial for developing and testing designs. Garcia-Haro *et al.* developed *ATMSWSIM*[43] (ATM SWitch SIMulator) as a discrete-event simulator (written in C), which could be used to evaluate different switch architectures and traffic patterns. Malgosa-Sanahuja *et al.*[78] further present an object-oriented development of ATMSWSIM. The use of an object-oriented approach (which is discussed in detail in Section 3.2.2.1 on page 27) was made to make the switch simulator more flexible, so that one piece of software could model *any* switch architecture. Brissinck and Dirkx[19] present a *platform-independent* simulation environment for analysing large switches. A software tool generates simulator code for different platforms (ie. sequential, parallel or dedicated hardware) from a single simulation model. The single simulation model only needs to be verified and validated once irrespective of the target architecture to be used for the simulation experiments. The tradeoff for the modeller is then between the simulation runtime and the cost of the machine used for the experiments.

To show how the use of dedicated hardware can improve the simulation runtimes of cell-level ATM network simulations, Stiliadis and Varma present the *FAST* (FPGA-based simulation testbed for ATM networks) system[100]. *Field Programmable Gate Array* (FPGA) hardware is used to model the operation of ATM cell-level switches to provide fast simulation of ATM networks.

2.5.3 The burst level

Pitts' *cell-rate* method[90, 91] is an example of a burst-level simulation technique specifically targeted at ATM network simulation. Data traffic is modelled by bursts of cells, where a burst describes a group of cells with the same cell transmission *rate*. Multiplexing of individual bursts is performed by queues which alter the cell rates of bursts traversing them to reflect the status of the queue. Cell loss and cell delay are determined by applying a set of equations to the instantaneous input, output and queueing rates for each burst in the queue. An event in this system marks a change in the cell rate of an individual cell stream. To avoid redundant processing effort, a two dimensional event list is used to ensure that concurrent cell rate change events are processed at the same time. This ensures that the output and queueing rates accurately reflect the current input cell rates

to the queue at each event. The arithmetic used in the system is floating point as cell transmission rates are used, and the number of cells in each burst is approximated as a floating point quantity (ie. fractional cell counts are permitted). A parallel[15] version, and a simplified version which ignores the previous state of the queue when an input change event is processed have also been developed.

Nikolaidis *et al.*[86] present a parallel burst-level simulation model of a high speed ATM multiplexer. Modelling at the burst level (where each burst represents a group of cells) is used to limit the number of events which need to be processed in the system. Rather than use a conservative or optimistic parallel discrete event system, a *time-parallel* technique is presented, where each LP in the system is responsible for simulating a separate time interval for the whole model. The argument for using this approach is that the poor spatial decomposition possible for a multiplexer model makes it unsuitable for TimeWarp-type parallel processing. A further development of this work for simulating entire ATM network models is presented by Akyildiz *et al.*[3]. Once again the burst level of abstraction is used along with a time-parallel simulation technique. The technique is stated as being unsuitable for network models with feedback, but can be used to speed up simulations of end-to-end models.

2.5.4 The call level

The same group which developed the cell-level CLASS simulator has developed a call-level simulator called *ANCLES* (ATM Networks Call LEvel Simulator)[80]. ANCLES is designed so that experiments comparing different routing algorithms and connection admission control (CAC) schemes can be performed. *Users* drive each call-level simulation, sending connection requests to the network, and acting as receivers for calls from the network. ANCLES has been designed to work in tandem with the CLASS simulator as a *hierarchical* call and cell-level simulation tool. The idea is the that the ANCLES simulation tool is used firstly to identify interesting behaviour in the network model which can then be studied in detail with the cell-level CLASS simulator. The process is automated, with the state of the ANCLES simulation preserved when control is passed to the CLASS simulator. Numerical result feedback from the CLASS simulation is then passed back into the ANCLES model, which continues execution until another interesting scenario is found and passed on to CLASS. Statistics gathering is also automated, allowing for models to be simulated at both levels until the user-specified level of confidence in the results has been reached. An example of the use of ANCLES to examine an adaptive routing algorithm for best-effort traffic in a network is

presented by Casetti *et al.*[20].

2.5.5 General techniques for improving ATM simulation performance

Various statistical techniques can be used with ATM network simulations in an attempt to reduce the runtimes necessary to observe *rare events* (eg. cell loss in a buffer). The use of traditional *Monte Carlo* methods to gain statistical confidence, when measuring parameters which have low probabilities of occurrence, requires many repetitions of lengthy simulation runs.

One means of reducing simulation runtimes is the *RESTART*[111, 110] (Repetitive Simulation Trials After Reaching Thresholds) method. RESTART works by running repeated simulation experiments to measure rare events from a point where the simulation has reached a selected threshold. For example, the rare occurrence of cell loss can be linked to the occupancy of a queue reaching a certain threshold level. When the simulation achieves this threshold, the state of the simulation is saved so that repeated simulations can be started from this point. This increases the relative occurrence of the rare event to obtain a tighter confidence interval for estimating its probability. Naldi and Calonico[84] compare the RESTART method against the *GEVT* (Generalised Extreme Value Theory) technique for estimating the probability of buffer occupancy in an ATM switch model. GEVT is a means of estimating the tail of a probability distribution of a random variable of interest (in this case buffer occupancy). The conclusion is that both techniques can offer simulation speedups, but RESTART is less critical as to the proper choice of simulation parameters. Neither technique relies on the characteristics of the computing platform used and so they can be used in any simulation environment.

Another technique is *importance sampling* which is a means of *biasing* the underlying probability mass used in a simulation in such a way that the rare events occur much more frequently. To correct for this modification, the results are weighted in a way that yields a statistically unbiased estimator. Care must be taken to ensure that the biased probability mass does not exclude the occurrence of the rare event of interest. The use of importance sampling also does not rely on the computing platform used, and so can be ported to any simulation environment. An example of the application of importance sampling for estimating the blocking probability of an ATM switch is given by Devetsikiotis and Townsend[34]. An introduction to various importance sampling techniques is given by Townsend *et al.*[104].

Simulations can also make use of mathematical models in what are known as *hybrid* simulators. One problem with analytical models is the suitable choice of parameters so that the model best represents the system under consideration. A hybrid model works by specifying a mathematical model of the system, and uses statistics produced by a simulation of the model to set the parameters of the analytical model. The aim of the technique is to gain results from the directly parameterised analytical model which would be computationally expensive to obtain from a pure simulation approach. As an example of this, Ani and Halsall[6] present a hybrid simulation model which can be used to predict cell loss rates in an ATM switch buffer.

2.6 Summary of chapter

The aim of this chapter was to give a general overview of simulation and modelling, with emphasis placed on the simulation of communication networks. The important techniques of mathematical analysis, discrete event simulation and parallel discrete event simulation were introduced. Each has found broad application in the modelling and simulation of communication networks.

A brief overview of cell-based ATM networking was given, as ATM technology has proven to be important for the provision of broadband networking infrastructure. Much research interest has been focused on the development of ATM technology and protocols, and this work has spawned many techniques for the efficient simulation of the technology. With this in mind, an overview was given of the different levels of detail which can be used to simulate ATM-like cell-based networks. Examples of the tools and techniques developed to provide efficient simulation at each level were then introduced and described. Finally, a brief overview of general statistically-motivated techniques, which can be applied to any simulator to help reduce simulation runtimes, was presented.

Chapter 3

Introduction to integer-time burst-level simulation

3.1 Introduction

The purpose of this chapter is to present the simulation techniques developed for integer-time burst-level simulation, and to outline the design of the fundamental building blocks required. The design and implementation methodologies used to provide a basic simulation framework are described first, as the choices made have an important bearing on virtually every aspect of the final simulation environment produced. Of particular importance is the choice of an integer variable clock, and the use of integer arithmetic throughout the simulation environment. The issues behind this choice are presented in this chapter.

The notion of a burst as used in this thesis is described next, along with the associated assumptions used when designing and implementing the rest of the simulator. The techniques required for the fundamental tasks of burst multiplexing, burst demultiplexing, burst queueing and burst switching are presented, along with the implementation details of the simulator objects which model these behaviours.

Finally, a summary of the work detailed in this chapter is presented.

3.2 A simulation framework

3.2.1 Choosing an implementation system

As the field of computer communications research is highly active, many simulation techniques, environments and languages have been produced. This is in addition to the "traditional" general purpose simulation languages such as *SIMULA*[12]. The advantage of using an existing simulation environment (or

language) is that the modeller can focus on the design of the model in question, rather than having to worry about coding the simulation support primitives required. In the case where the modeller produces their own primitives, they must be carefully checked to ensure they function correctly. When using an existing package, ideas can be experimented with quickly and relatively easily to aid refinement of the model under development. The disadvantage of this approach is that the modeller is at the mercy of whatever assumptions the designer of the simulation environment made, as well as perhaps non-optimal performance due to the use of a general tool for a specialised model.

3.2.1.1 Moving from a general to a bespoke simulator

With the pros and cons of using an existing simulation environment in mind, SIMULA was chosen as the initial tool with which to design the burst-level simulator[29]. Although SIMULA itself provides sufficient simulation primitives to create models, it leaves many of the implementation issues with the modeller. Franta[40] and Mitrani[83] each present SIMULA-based simulation frameworks, although the DEMOS[11] simulation framework was chosen in preference. *DEMOS* (Discrete Event Modelling On SIMULA) provides an object-oriented set of process-based simulation primitives built on top of SIMULA. Bratley *et al.*[16] compare and contrast various simulation environments and DEMOS is declared as one of the best options for small-scale simulations. However, they strongly advocate the use of bespoke developments for larger scale simulations to ensure that the modeller knows exactly what is happening in the simulation at every level.

The first generation of simulation objects was produced using the DEMOS system as the simulation environment. As the work progressed, the set of simulation primitives necessary for the simulator could be narrowed down to the optimal number. As an aim of the project was to produce an efficient simulator, a prototype *C++*[101] simulator was produced which mirrored the actions of the DEMOS-based simulator, so that the two could be compared. It was found that the C++ version of the simulator could execute a much greater number of events per second than the DEMOS system, suggesting that the low overheads of a bespoke simulation environment coupled with a good C++ compiler delivered immediate benefits over the DEMOS and SIMULA compiler combination. With this in mind, all subsequent development of the simulation environment and burst-level simulation objects was moved to the C++ platform. Any performance enhancing measures highlighted during development could then be applied to the

simulation environment which was optimal for the simulator being produced. The DEMOS system is a general purpose simulation language which makes it difficult to customise to ensure that the simulator is as efficient as possible for the model being investigated.

The C++ language was chosen for the final implementation of the simulation environment because of the abundance of good compilers and debugging tools available on all of the popular computing platforms. Tools to perform runtime checking (ie. to trap reads of uninitialised variables) and memory leaks (ie. where dynamically allocated storage is not returned to the free memory pool) were used at every stage of development to help ensure the robustness of the simulator produced. Sanderson and Rose[94] present an example of how the C++ class structure can be used to model a simple computer system. On a larger scale, Mikler, Wong and Honavar[82] describe the design of a simulator for large communication networks implemented in C++.

Other tools, based on C++, are available such as *Sim++*[26] (as previously used by the author[28]) and *C++SIM*[75] which mimics the simulation primitives provided by SIMULA. However, the use of "vanilla" C++ ensured that the environment produced was not compromised by relying on any particular proprietary systems. Full portability of the simulator code between different computer platforms (equipped with a suitable compiler) was also ensured. With a completely custom environment, any simulation optimisations presented by the choice of techniques could be made to the simulation framework with relative ease.

Other object-oriented languages have been shown to be suitable for discrete event simulation. Kreutzer[71] describes how the *Beta* language, designed as a successor to SIMULA, can be used for discrete event simulation. A discrete event simulation package, *SimJava*[57], built using Sun's *Java* language is described by Howell and McNab. SimJava allows simulation models to be executed over the World Wide Web with any Java-compatible browser.

3.2.2 Design and Implementation Methodology

3.2.2.1 Using an object-oriented approach

It was decided early on to use an object-oriented (OO) design approach in the simulator development. Communication network elements such as users, switches, multiplexers and queues can be designed as objects which can be arbitrarily connected in the same way as the real networks being modelled. An object-oriented approach provides the benefits of encapsulation, inheritance, polymorphism and scalability. Encapsulation allows for the code and data structures of each network

element object to be clearly defined with an appropriate interface. As long as the interface remains constant, the behaviour of the element under consideration can be changed with minimal disruption to the rest of the simulator. Inheritance allows for network elements with similar behaviours to be quickly and efficiently modelled as sub-classes of generic types. For example, different classes of "user" object could all derive from one generic "user" containing the necessary interface logic for the other objects in the simulator. Polymorphism allows objects to re-define inherited behaviour and be altered without affecting the object interface. Consistent object interfaces allow for arbitrarily sized simulation models to be produced easily (ie. scalability) by just adding more objects.

Kosbar and Schneider[70] introduce the OO design model and argue that it increases the flexibility and extensibility of simulation packages. Joines[65] also recommends object-oriented simulation modelling over the more traditional procedural style. The behaviour of each network object can be verified in small scale experiments before large experiments are constructed from the same objects. Well designed object interfaces allow for the straightforward addition of new features to a system and the easy modification of existing objects. It would be possible, for example, for a library of different user traffic profiles to be created (eg. one for real-time video, one for FTP transfer, etc.) which could then be appropriately "plugged" into the simulation model.

The one drawback of using an object-oriented approach is the extra overhead introduced into the executable program. This performs the runtime dynamic binding necessary to manage dynamic objects, and to ensure the correct methods are executed depending on the object type. The first C++ compilers were written to translate C++ code into C which could then be compiled with a standard C compiler. Modern C++ compilers produce native code without the need to produce any intermediate code. C++ compiler writers have also ensured that the overhead of the runtime dynamic management of objects is as small as possible. The implicit overhead is one of the tradeoffs for the inherent advantages of using an object-oriented language for simulation studies.

3.2.2.2 Process-based modelling

A *process-based* approach is used in the simulator design due the merits of this technique when modelling systems. As discussed by MacDougall[77], the main advantage is that simulation programs written in process-based languages can be constructed as straightforward descriptions of actual system operation. The use of an event-based language would lead to complications with the requirement

that each event often contains actions for more than one system component. This makes the construction of large scale simulations difficult especially when one requirement is the need to model arbitrary networks. Process-based languages offer a hierarchical development framework such that the level of abstraction can be changed during development (ie. existing processes can be decomposed into *activities* which themselves may be represented by processes). The similitude of model and system makes it easier to ensure that the model is a valid representation of the system, particularly in a development environment where the system design is undergoing constant change. A process-based approach is well suited to object-oriented languages as an object can represent a process. Process-based simulations containing more than one interconnected process require some means of communication to trigger possible changes of state. Discrete events are used for this purpose as a discrete event, scheduled for a process at some simulation time, indicates that some action is required in the process.

In terms of performance, the use of a process-based approach can introduce extra overhead into a simulation. In a network simulation, for example, each node in the network will need to be represented by a process, which could be an object instantiation in C++ for example. Each object requires some physical memory in the computer running the simulation, and the more objects simulated, the greater the requirement. Also, as the means of progressing each process is through the use of discrete events, the greater the number of processes, the greater the pressure on the event list management scheme used which has to schedule and process the events generated. In process-based simulation, when an object processes an event a *context-switch* will typically occur in the simulation runtime system to activate the object. Each time control changes to a different object (ie. process) then some overhead will be incurred leading to increased runtimes. Event-based techniques[77] differ in that each event explicitly progresses the state of the entire simulation. A simulation run executes a list of time ordered events of this kind. Each event can be modelled as a *function* and so no context switches are required. This means that event-based techniques have low execution overheads. The disadvantage of an event-based technique is the difficulty in producing the events necessary to perform the required actions for the whole model under simulation. This is especially difficult when large-scale simulations studies are required. Huang and Iyer[58] have shown that simple C++ process-based models can be transformed into event-based models by using compiler optimisation techniques.

One approach to simulation program implementation is to use some means of program generator which can generate simulation code for a model. Some

higher-level description of the model, in the form of a language or graphical representation, is used by the program generator to generate the code used to simulate the model. One advantage of this technique is that the modeller need only learn the higher-level model design language rather than the underlying operation of the actual simulation language or environment used to run the simulations. Another advantage is that the simulation program generator may produce code for several different simulation language targets, even allowing for different simulation methodologies to be used to simulate the same model. For example, Mathewson[81] presents a simulation program generator which can produce object-oriented, activity or event-driven simulation implementations from a higher-level diagrammatic entity-cycle description of the simulation model.

3.2.2.3 Choosing an object-oriented process-based development environment

The first development tool used, DEMOS, provides an object-oriented process-based simulation environment. DEMOS is a *pure* process-based system as SIMULA provides *co-routine* support (ie. the ability of objects to possess autonomous actions and to suspend and resume these at desired points). This allows DEMOS *entities* (the DEMOS name for objects in a simulation) to use looping and conditional branches within themselves and be able to suspend and resume as required. Certain compromises are necessary for other object-oriented languages when they are used for process-based discrete event simulation. Pooley[92] describes a particular approach, which uses the `switch...case` structure in C++, to perform process-based discrete event simulation without intrinsic co-routine support being available. One technique commonly used to provide co-routine support in C++ is through the use of *threads* (lightweight concurrent processes). Sim++ and C++SIM, for example, make use of threads in their operation. Thread packages are quite often architecture or operating system dependent, which reduces the portability of any code produced.

The C++ simulation environment implemented in this project uses a discrete event driven process-based approach. Objects are scheduled for execution when they receive an event, and in turn can schedule events for other entities. No attempt has been made to mimic co-routine support, which could be made available through the use of a thread package. Instead, each object must hold information about its state such that the arrival of a new event can be handled correctly. The events scheduled for each simulation object carry a *message* payload for that object. The "message" is what is actually processed by the object, and the term

is used interchangeably with "event" throughout the rest of this chapter.

3.2.3 Choosing integer time based simulation

3.2.3.1 Why use integer variables for simulation time?

Traditionally, simulation languages have been written to use floating point variables to represent the simulation time. DEMOS is an example of this approach. While floating point numbers are convenient for mapping a simulation to "real-world" timings, they present a bottleneck when designing large simulations. For instance, when modelling high speed communication links, the precision of floating point numbers when representing time periods becomes a problem. To help preserve accuracy, variables are computed to `double` precision (normally a 64-bit quantity) and such floating point arithmetic can slow down such simulations, even if the accuracy obtainable is sufficient. A problem with the initial DEMOS burst-level simulator was that of floating point precision. When calculating the number of cells in a burst, rounding of the result could lead to incorrect cell counts being passed between various entities. When simulating higher communications speeds, the problems with precision worsened. Rounding problems can be minimised, but this adds an extra level of complexity to the approach.

The alternative to floating point arithmetic is to use an all-integer approach in both the simulation primitives and the simulation objects themselves. Modern superscalar microprocessors provide a number of fast integer and floating point arithmetic execution units and the ability to schedule several instructions per clock cycle to give high performance. However, each instruction still has an associated cost in terms of result *latency* and *throughput*. The result latency is the time in clock cycles from instruction issue to the result being available. Instruction throughput is defined as the delay in clock cycles before the result from the next instruction of that type (if scheduled directly after the first instruction) is available. Table 3.1 gives the typical cost of arithmetic instructions for two popular processor cores for 32-bit arguments (sources [5] and [60]).

As can be seen from Table 3.1, integer arithmetic operations are intrinsically faster than their floating point counterparts. Hence, the use of an integer global clock will offers an immediate benefit. This may immediately improve the performance of global event list handling in a discrete event simulator, as a great deal of arithmetic is performed using variables representing time. As the simulation primitives are often the most frequently executed parts of a simulation program, it is advantageous to make their execution as efficient as possible. The requirement to use an integer variable to represent simulation time was another reason

Processor	Instruction	Latency	Throughput
AMD K6-2/3	Integer add/sub	1	1
	Integer mult	2-3	2-3
	Integer shift	1	1
	Integer divide	≈ 18	not pipelined
	FP add/sub	2	2
	FP mult	2	2
	FP divide	34 (double)	not pipelined
Intel Pentium II	Integer add/sub	1	1
	Integer mult	4	1
	Integer shift	1	1
	Integer divide	≈ 34	not pipelined
	FP add/sub	3	1
	FP mult	5	2
	FP divide	36 (double)	not pipelined

Table 3.1: Instruction latency and throughputs for two common microprocessor cores

to move the simulator development from DEMOS to C++, as DEMOS is written to treat time as a floating point quantity.

3.2.3.2 Representing time with an integer variable

The immediate issue with integer time is to decide just how it represents real time. The original approach for the simulator (as described in [30] and [31]) was to have one unit of integer time represent the time taken by the fastest simulated communications link in the model to transmit one cell. This is a reasonable simplification, as data transmission rates in high speed networks tend to be close to integer multiples of one another, thus a wide range of network transmission speeds can be simulated. A similar simplification is made in both the CLASS[1] and YATS[108] simulators. However, allocating more than one unit of integer time to the fastest transmission time is more flexible especially when considering the needs of burst multiplexing as described in Section 3.4. If an integer clock is used some means of translating this to real time is necessary. This is easily achieved through the multiplication of some floating point constant. Such a translation could be achieved with a floating point clock, but one would miss out on the intrinsic performance advantage of using an integer clock in the first place. Decoupling the simulation clock from "real" time through the multiplication of a constant allows for ever increasing transmission speeds to be simulated with no loss of accuracy. Slower transmission times can be simulated by some multiple number of units of the simulation clock. One consideration is the maxi-

mum simulation time afforded by the use of an integer clock. If, for example, the fastest link simulated is 622Mbs^{-1}, and 10 units of the clock are used to model the transmission time of one cell at this rate, an unsigned 64-bit precision variable (eg. type `unsigned long long int` in C++) representing the clock will give a maximum simulation time of approximately 39,000 years.

With the global simulation time represented by an integer variable, it is intuitive to use integer arithmetic throughout the simulator. The accuracy of integer arithmetic is assured when compared with floating point arithmetic, and this is of benefit when producing network objects for the simulator as described in the following sections. Network simulations, especially when using a discrete event approach, will have very many events on the global event list and efficient methods of handling these events are important. If all quantities are integer, the storage requirements for each event in the simulator can be streamlined as far as possible. This enables a greater number of events to be stored in the physical memory of the computer running the simulation, with obvious performance benefits.

3.2.4 Driving the simulation framework

This section presents the object structures chosen for the implementation of the C++ burst-level simulator. One of the advantages of an object-oriented approach is that the functionality of core objects may be changed at any time as long as the interface remains constant. Even fundamental features, such as the event list management technique, can be changed if required.

3.2.4.1 The global event list

The most crucial feature of a discrete event simulator is the global event list. Events are placed in order of increasing simulation time into the list, while the event with the lowest simulation time represents the current global time of the simulation. As well as the correct ordering of events being essential, a choice has to be made as to how events scheduled for equal simulation times are handled. A common strategy is to insert such events into the global event list in a strict "first in, first out" (FIFO) order such that the events are processed in the same order as their insertion into the list. Such an approach was taken, with the strict ordering assumed when designing the simulation objects.

From an implementation point of view, representing the global event list with a doubly linked list is particularly easy using C++. An example of this is shown in Figure 3.1. The head of the list represents the current event (and hence the current global simulation time) while the tail of the list represents the final event

scheduled thus far. The insertion of a new event (or the rescheduling of a current event) is performed by scanning the list until the insertion point is found. The pointers of the new event, as well as those of the events both preceding and succeeding the insertion point, can then be changed to incorporate the new event. Strict FIFO ordering of events is secured by ensuring that the simulation time of the preceding event is less than or equal to that of the new event, whereas the simulation time of the succeeding event is greater than that of the new event. Processing the current event is simply a matter of removing the event at the head of the list and promoting the successor event to be the list head.

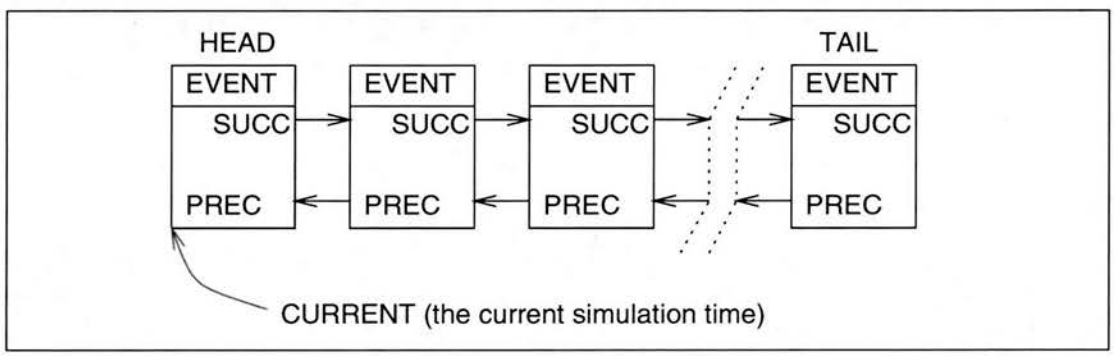


Figure 3.1: Representing the global event list as a doubly linked list

The major drawback of implementing the event list with a doubly linked list is that event insertion may become a costly operation when the number of events described by the list is large. In the worst case, the time complexity of the insertion algorithm will be $O(N)$ where N is the number of events in the list. There is some flexibility in how one scans for the correct insertion point, as this can be from either the head, or the tail, of the list. Scanning from the head may be preferable when the difference between the simulation time of the new event and the current event is small. Scanning from the tail of the list may be preferable for larger simulation time differences, as the assumption is that there will be a greater time density of events near the current event.

The problem of poor performance of doubly linked event lists for large numbers of events has been known for some time. Vaucher and Duval[109] present an overview of simulation event list algorithms, suggesting the use of various indexed event list and binary tree algorithms as a means of improving event list performance. Wyman[114] also presents the indexed event list technique, an example of which is shown in Figure 3.2. In this technique, a separate list of time ordered event "keys" which point to events in the event list is maintained. The main event list is still a sequential time-ordered doubly linked list of events. When a

new event is to be inserted into the list, the index is searched first to find the last key pointing to an event with a simulation time less than that of the new event. The event list is then sequentially searched from this point until the insertion point is found. The difficulty with this technique is maintaining the list of index keys such that the number of events described in the time epoch represented by two adjacent index keys is less than some maximum number. There are several strategies for maintaining the key index which directly influence the worst case time complexity of this technique. If, for example, the number of keys, K, can vary such that the maximum number of events between adjacent keys, M, is kept constant then the worst case time complexity of event insertion is $O(M+K)$. The number of keys, K, is N/M and the worst case time complexity of event insertion is minimised if $M = \sqrt{N}$ to become $O(\sqrt{N})$. Franta and Maly[41] introduce a two level indexed list, where a secondary list of keys pointing to the primary key index is also maintained. Here, the number of secondary keys remains constant, whereas the number of primary keys can change to ensure the number of event notices represented by adjacent primary keys does not exceed M. The worst case time complexity of event insertion is again $O(\sqrt{N})$ but this does not take into account the work performed during index key management.

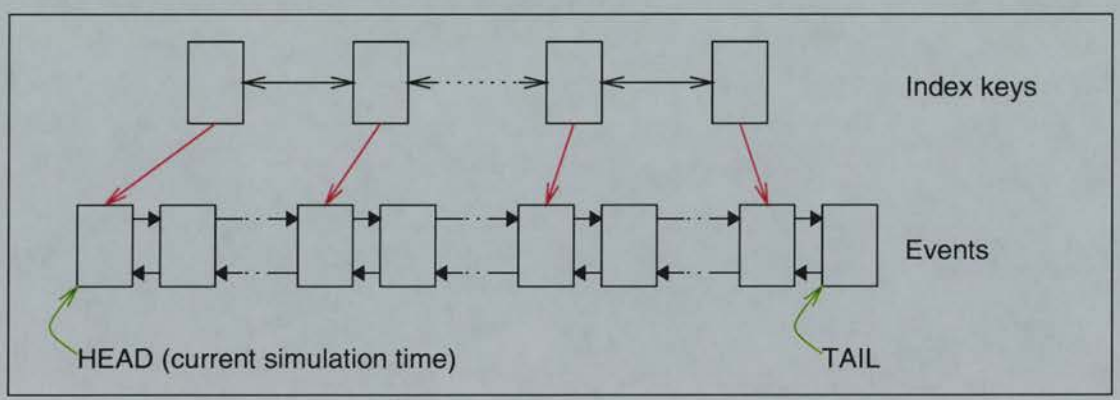


Figure 3.2: An indexed doubly linked global event list

Vaucher and Duval[109] also present the use of binary trees as a means of improving the efficiency of event list operations. Knuth[69] and Cormen[25] describe binary trees and show how the worst case time complexity of basic tree operations is $O(\log_2 N)$. A small binary tree is shown in Figure 3.3. Each event is represented as either a root, branch or leaf node in the tree with the current event being mapped to the node with the lowest simulation time. There are several approaches for determining the method of node insertion into the tree, but an important property is that the tree must preserve the strict FIFO ordering of events. The use of *post-order* and *end-order* tree algorithms is presented by

Vaucher and Duval as a means of preserving event ordering. The post-order tree is a binary tree where left hand branches represent event notices with times less than or equal to that of the root, whereas right hand branches represent times greater than that of the root. Traversal of the tree to produce a time ordered sequence of events is left branch, root and then right branch. FIFO event ordering is preserved by inserting an event with an equal time to an existing node in the place of the existing node, which in turn is pushed down the left branch of the new node. Any right branch of the original node is moved to be the right branch of the new node.

The end-order tree is also a binary tree having the property of producing a stack of nodes when their simulation times are equal, so that strict FIFO processing of the events is ensured. The simulation time at a given node is greater than or equal to the simulation time of any node in its subtrees. This requires that tree traversal for event processing is in the order left branch, right branch and then the node itself. The advantage of this is that an event can be placed fairly high in the tree without the need to follow a branch to the end to find an empty branch as in the post-ordered tree. This helps reduce the overhead of event insertion into the tree.

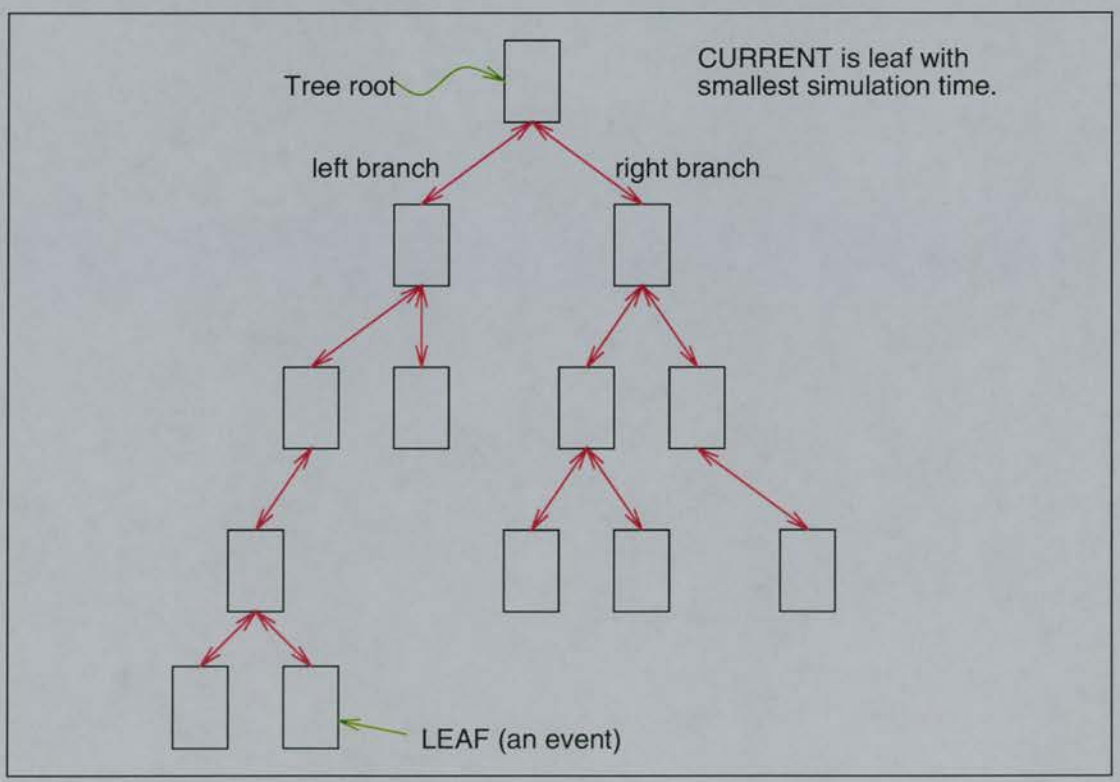


Figure 3.3: A binary tree representation of the event list

Mitrani[83] compares and contrasts the indexed list versus the binary tree

as a means of improving event list efficiency. He concludes that the binary tree approach may be more efficient for large numbers of events (where event insertion is concerned), whereas the indexed list may well be better for smaller numbers of events. Vaucher and Duval conclude that the distribution of simulation times for the event notices to be inserted into the list, be it an indexed list or a tree, determines the relative performance of each technique.

Although binary tree structures may give excellent performance when inserting event nodes, finding the next event to process requires a fresh search of the tree each time in the case of the end-ordered tree. This is inefficient when compared to the doubly linked list, where the next event to process is always at the list head. In a typical simulation, events will be dynamically created (when scheduled) and be destroyed (when processed) and structures such as the end-ordered tree are inflexible under such circumstances. This is because events may potentially be processed out of time order, due to a left branch being removed while events with smaller simulation times exist in the right branch. When the event insertion routine spots an empty left branch, this may be used rather than checking for the presence of events with smaller simulation times in the right branch. The post-ordered tree fairs better when dealing with dynamic event insertion and removal, as the position of the next event to process is known after execution of the current event.

To help improve the efficiency of the event list operations, both an indexed event list and a post-ordered tree have been implemented in the C++ simulator. The indexed event list requires relatively little alteration to the basic doubly linked list. The post-ordered tree technique requires a complete replacement of the event list methods. However, the object interface of the event list is unaltered, regardless of the event list management used.

3.2.4.2 Indexed event and post-ordered tree event list comparison

The indexed event list implemented in the simulator was designed to allow the number of index keys to vary, while preserving a maximum number of events between adjacent indices. The event list dynamically manages the doubly linked list of event indices, adding and removing individual index entries as required. When a new global event list is created, an initial event index is placed at the maximum simulation time. This ensures that there is at least one index to allow other indices to be added relative to it. Adding an event to the global event list involves searching the event index from its head (ie. the event index with the lowest simulation time) until the first index with a time greater than the time of

the event to be added is found. The global event list is then searched backwards (ie. in simulation time), from the event pointed to by this index, until the correct event insertion point is found. Searching in this manner ensures that a strict FIFO ordering is preserved among events with equal simulation times. After the event has been added, an event counter is incremented in the event index used (the *current* index). If this event count is greater than the maximum permitted, a new index is created and inserted between the current index and its succeeding index. The event count for the current index is then set to the maximum number of events, while the event counter for the new index is set to the previous count minus the maximum number permitted. The difference in the counts may be greater than one if removal of events from the event list (during a simulation lifetime) causes event indices to be reassigned to other events. The current event index then points to the event preceding the event it originally pointed to. The new event index points to the original event.

The next step is to "gather" the event indices to the right of the new index (ie. in order of increasing time). This involves removal of adjacent succeeding event indices and adding their event counts to the event count of the new index. This continues until the event count would be greater than the maximum permitted if the next event index were added to it. If no events were dynamically removed from the event list during a simulation run there would only be one such index to the right of the new index (ie. the index gather would simply examine the right hand index). However, event removal affects the number of events per event index, so that there could be more than one index to be gathered due to index counts changing. At the end of the index gather, the event pointer in the new index is altered to point to the event from the last event index added to it. This "index gather" step is optional, but the extra work needed will be justified if the number of index events is reduced, and subsequently the cost of event insertion is reduced also.

A post-order tree event list class was also produced in the simulator. The post-order tree was structured with left branches of an event node having simulation times less than or equal to the node, whereas the right branches have simulation times greater than that of the node. Insertion of an event involves traversing the tree from the tree root until the suitable insertion point is found (ie. a suitable null branch pointer at a node). The behaviour is altered when the event to insert has a simulation time equal to that of an event currently in the tree. The insertion point, in this case, is the location of the existing event. To preserve FIFO event ordering, the existing event is made the left branch of the new node, thus forming

a FIFO left branch "chain" of events with equal simulation times. To preserve the structure of the tree, if the existing event had a right branch, this is moved to become the right branch of the inserted event. When a simulation starts, the first event to process is found by traversing the left branches in the tree until the smallest simulation time is found. After the first event has been processed, the current event pointer can be moved to the next event to process without the need for a whole tree traversal from the root. To ensure that the current event is not considered when processing the event, the current event is removed from the tree and the links in the tree updated to reflect this. This is necessary, as a simulation object scheduled by the current event may itself schedule events which have to be inserted into the tree. With the current event removed, the new event (or events) will be inserted into their correct positions.

In order to compare the relative performance of the different event list handling techniques, a simple event insertion test was performed using each structure. The test results should only be used as a guide to potential performance, rather than as an absolute performance indicator. As Vaucher and Duval[109] found, the *distribution* of the order of insertion of events, based on their simulation times, has a great bearing on the performance of whichever event list mechanism is used. Other factors, such as the pointer manipulation required by each technique to perform dynamic event insertion and removal, also need to be considered. The synthetic test bed used for the test may also benefit from effects such as processor cache locality of the event structures which will influence performance. Real simulation runs will have rather more work to do, as the code for many simulation objects will be executed in addition to the simulation primitives.

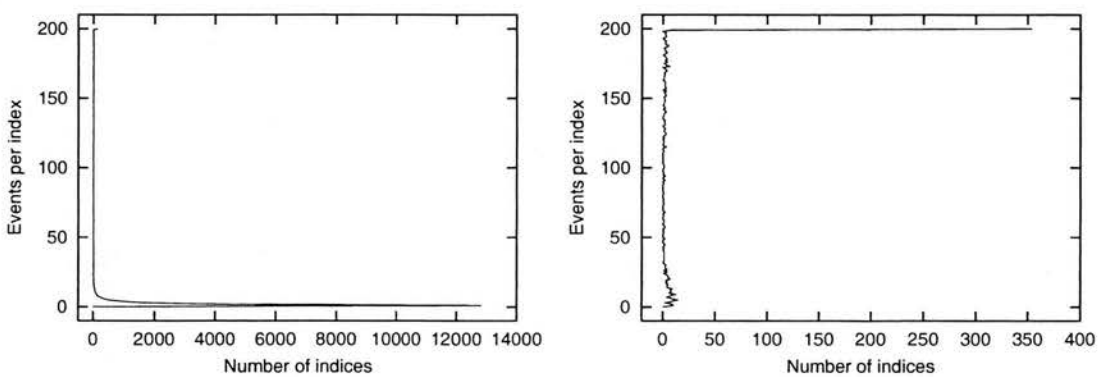
In the tests, 100000 events were inserted into an event list. The simulation time for each event was drawn from a random uniform integer distribution which produced values in the range of 0 to 1000000. All of the tests were performed on the same workstation which was unloaded by other processes. Table 3.2 gives the runtimes, in seconds, of the the event insertion tests for each technique. As a reference, the standard doubly linked list was included along with the indexed event list (with and without event gathering) and the post-order tree. The maximum number of events per event index for both indexed list algorithms was 200.

As can be seen from Table 3.2, the post-ordered tree gives the best performance for this basic example. The simple doubly linked list has the worst performance, as would be expected. It is clear from the results that the use of event index gathering, after the creation of a new index, is a justifiable overhead in terms of the performance improvement available. The number of event notices

Experiment	Runtime (secs)	No. event indices
Simple Doubly linked list	560.3	n/a
Indexed list	38.7	21533
Indexed list (with index gather)	2.8	810
Post-ordered tree	0.5	n/a

Table 3.2: Program runtimes for the different event insertion techniques

is greatly decreased when event gathering is used (as would be expected) and the performance improves accordingly. Figure 3.4 shows the distribution of the number of events described by each event index when event index gathering was both enabled and disabled. It is clear where the performance penalty of not using event index gathering lies, as the majority of the large number of event indices produced describe a small number of events. Event insertion, under these circumstances, will be penalised by having to search many event indices to find the relevant insertion point for the event.



(a) Without event index gathering

(b) With event index gathering

Figure 3.4: The distribution of events to event indices after 100000 event insertions when event index gathering is either used or not

The simple event insertion test used an event count maximum of 200 for each event index algorithm. Figure 3.5 shows the range of runtimes for the simple event insertion test for an indexed event list with event gathering, versus the maximum number of events for each event index. As can be seen, the performance is best when the maximum number of events is in the range 200 to 300 which is approximately the square root of the number of events inserted, which was 100000. This is approximately the situation predicted by Mitrani [83], as the case where best performance may be seen.

It is clear from the tests that the indexed event list and post-order trees offer

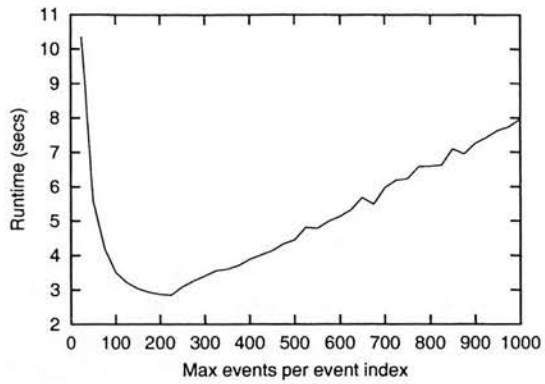


Figure 3.5: The effect of maximum event count on runtime performance for the index event list insertion algorithm

large performance benefits over the simple doubly linked list when it comes to event insertion. However, as this was a simple test not necessarily representative of a real simulation, both the indexed event list (with index gathering) and the post-order tree event list methods were implemented in the simulator. This was so comparative performance evaluation of the event list structures could be attempted for real simulation runs.

3.2.5 Why simulate at the burst level?

As described in Chapter 2, there is a choice for the level of abstraction when modelling high bandwidth communication networks. The most detailed level of abstraction is at the cell level (in a cell-based network such as ATM), where the passage of each and every cell through the "real" network is modelled. Although this is the technique capable of producing the greatest accuracy, the penalty is that simulation runs require a vast amount of compute time. The next level is the burst level, where cells sharing some property are grouped together and the burst is the entity traced though the simulation. Simulation at the burst level has the potential for reducing the event count necessary to perform a simulation when compared to an equivalent simulation at the cell level. This has the advantage of reducing the compute time at the expense of the accuracy obtainable (cf. the cell level). The highest level of abstraction is the call level, where the entire duration of a data connection to the network is modelled. The call level can model large amounts of simulated time within reasonable compute times. The disadvantage is that producing models relating the behaviour at the call level with the behaviour at the cell level is difficult.

It was decided to base the simulator produced in this project on the burst level of abstraction. If the aim is to produce an efficient means of simulating

large amounts of network time, the cell level is too detailed, whereas the call level is too abstract. Capitalising on the reduced event count for simulation at the burst level, the goal is to produce algorithms and techniques which minimise the cost of abstracting to the burst level such that it outperforms the equivalent cell level model. As the burst level is an abstracted level of detail, one cannot expect the results to be as accurate as those achieved at the cell level. However, if the accuracy penalty can be determined not to be overly restrictive, and if the run times improve over cell-level simulation, then the technique may be useful. Finding the conditions under which the burst-level simulator performs poorly, as well as when it performs well, is essential to a prospective modeller wishing to use the technique.

3.2.6 Simulation objects

This section describes the basic object structure of the burst-level C++ simulator produced. Objects are represented in C++ by *classes* which encapsulate the data and methods for the object being described. Other classes may be derived from a class (known as the *base* class), such that the *derived* classes inherit the data and methods of the base class. Derived classes may redefine the base class methods if modified operation is required. A special case is where a method is declared as *virtual* in the base class, meaning that the method is only defined in the classes derived from that base class. The central classes to the simulator are `object`, `ev_list`, `event` and `burst` which are defined as follows:

`class object` is the base class from which all simulation objects in the simulator are derived. Class `object` contains methods to schedule objects of class `event` for a class `ev_list` object (specified when the object is instantiated). Processing of events scheduled for the object is handled by a virtual method (`object::process`), whose behaviour is defined by the specific implementation of the process method in each derived class.

`class ev_list` is the global event list handler for the simulation. Class `ev_list` maintains a sequential time ordered list of class `event` objects. Methods are provided for the insertion and removal of events for any simulation time greater than, or equal to, the current value of the global clock. The `ev_list` object also performs the simulation run by removing the `event` object at the head of the event list (ie. that with the smallest simulation time), setting the global clock to the time represented and then passing the event to the `process` method of the `object` for which it is scheduled. The simulation

run is terminated when the event list reaches a special `END_SIM` message, which is scheduled when the simulation is started for a user-defined end time.

`class event` is the class representing the discrete events in the simulator. Each `event` object may hold a pointer to a class `burst` object which is passed to the `object` for which the event is scheduled. The `event` object holds a pointer to the `object` to enable the `process` method to be called.

`class burst` is the class representing the bursts whose communication is being simulated. Bursts hold information detailing the contents and transmission speed of the data being represented by the burst.

Figure 3.6 shows the main relationships between the object classes in the simulator. The figure shows a simple doubly linked list for illustration purposes, but a post-ordered tree structure for the event list may also be used (as described in the previous section).

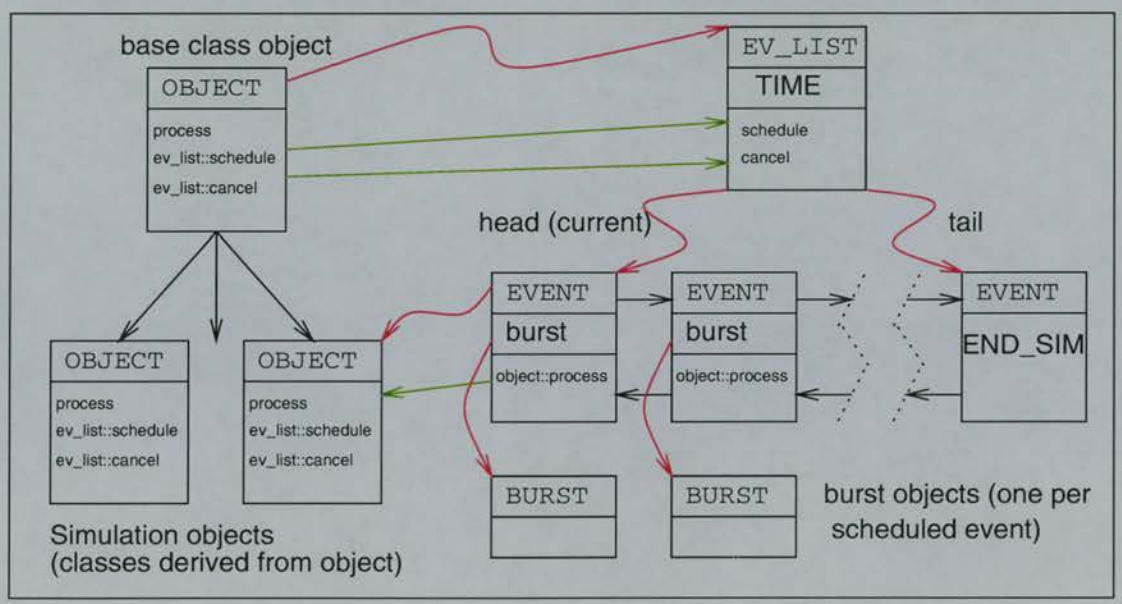


Figure 3.6: The object class relationships in the C++ simulator produced

Other essential services such as random number generation, and file logging are available to simulation objects through appropriate classes. The random number generator implemented in the simulator is a direct translation of the one provided in the DEMOS package. Simulation objects have to provide their own statistics gathering functions although several user-selectable levels of file logging are provided as standard in the `object` class. This allows the experimenter to choose varying levels of detail in their logging of the progress of a simulation

to allow for detailed behaviour checking. When the user is satisfied with the operation of their model, the level of file reporting can be reduced through a simple variable passed to each object. Of course, for maximum efficiency the logging code can be removed completely from each simulation object.

3.3 The Anatomy of a Burst

3.3.1 Defining a burst

Figure 3.7 shows how a burst is used to represent a group of cells in the simulator. As shown, a burst is defined as a stream of cells which have both equal transmission and equal interarrival times. A single burst is described by a pair of `START` and `FINISH` messages unique to that burst. Each `START` and `FINISH` message is represented in the simulator by an event. For example, a `START` event scheduled for an object is equivalent to the arrival of a `START` message at that object at the scheduled time of the event.

The time period spanned by the simulation time difference between the `START` and `FINISH` messages represents the sum of the physical time to transmit each cell in the burst plus the product of the interarrival time and the number of cells described. The simplification introduced, is to compute for an average cell the sum of the actual transmission time plus one inter-cell arrival time and call this the *averaged cell transmission time* (ACTT) for a cell in the burst. The ACTT for a burst is the period of simulation time required to elapse to simulate the successful transmission of one cell from the burst.

As a burst is an abstraction of the cell level, the definition of one ACTT being equal to exactly one cell transmission time plus one inter-cell arrival time is not necessarily rigid. Depending on the resolution (for accuracy purposes) required by the modeller, the ACTT chosen for a burst need only be the *average* for the constituent cells. Choosing the resolution of the burst sizes, in terms of cell numbers, will have implications for both the accuracy and runtime performance of the simulator. Too low a resolution (ie. large burst sizes) may average away the detailed behaviour of the system under consideration. On the other hand, too high a resolution (ie. small burst sizes) may impact performance due to the overhead of implementing the abstracted technique.

The burst shown in Figure 3.7 only describes one stream of cells being transmitted. When a greater number of streams need to be simulated as concurrently using the same transmission medium, the burst has to describe each of the concurrent streams. A cell multiplexer in an ATM network, for example, will merge

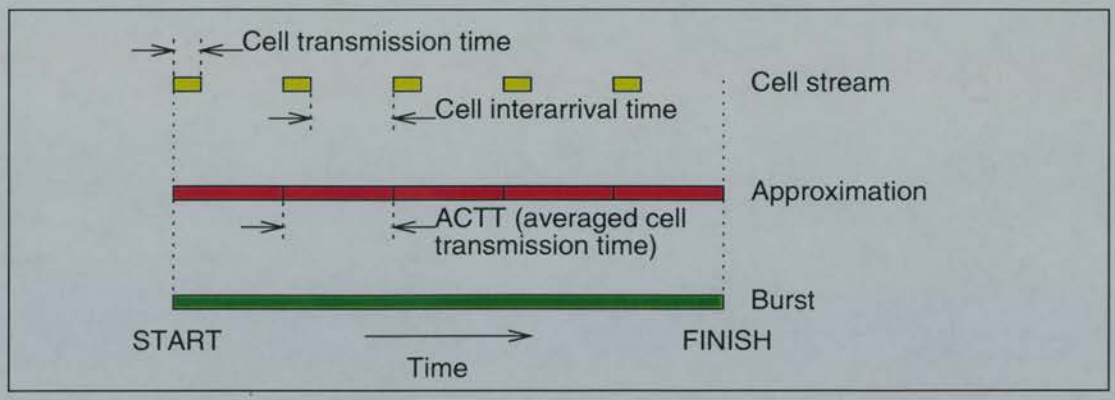


Figure 3.7: The relationship between a burst and the group of cells it represents

many input streams of cells to share the same transmission line at the output. To simulate this, the approach taken is to produce a single burst (ie. one pair of `START` and `FINISH` messages) which describes each of the concurrent streams at the output. This is achieved through the technique of *burst multiplexing* as described in Section 3.4. As a burst may describe one or more individual cell streams, the `START` and `FINISH` messages need to carry the information shown below.

The following information is carried in a burst `START` message:

- The averaged cell transmission time (ACTT) for the burst.
- The maximum number of cells which can be carried by the burst (the actual number carried is confirmed in the `FINISH` message for the burst).
- The number of cell streams of which the burst is comprised.
- Each component stream contains the following details:
  - The ACTT for that stream.
  - The maximum number of cells in that stream (the actual number is confirmed in the `FINISH` message for the burst).
  - A unique stream identifier.
  - A network source address for the stream.
  - A network destination address for the stream.

The following information is carried in a burst `FINISH` message:

- The total number of cells carried by the burst.
- The number of cell streams in the burst.

- Each component stream contains the following details:
  - The unique stream identifier.
  - The number of cells carried from the stream.

As can be seen from the information carried in the `START` message, a burst describing more than one cell stream contains both an ACTT for the entire burst as well as an ACTT for each stream carried. This is because the burst multiplexing process produces one ACTT value which is used to describe the entire burst. The idea is that a multiplexed set of streams will be sent serially along the communication medium which is being shared. This is equivalent to one stream of cells with an ACTT which reflects the number of cells transmitted in that timespan. The multiplexing process produces this single ACTT value for the burst from all of the concurrent streams being merged. An example is shown in Figure 3.8 where three streams are multiplexed. The multiplexed stream contains all of the cells from the component streams, with an ACTT calculated such that the total duration matches the concurrent duration of the three streams. Other simulation objects use the burst ACTT for further transmission of this burst. Individual streams are reclaimed when the burst is demultiplexed or switched (as each component stream may have a different destination address). Using the burst ACTT to describe the streams which have been merged abstracts the detail of the concurrent streams to a single stream. Each component stream can be demultiplexed when required as this information is not lost. A point to note is that cell ordering information of the component streams in the multiplexed burst, as shown in Figure 3.8, is **not** what is actually carried. Any calculation of individual stream cell allocations by other simulation objects on the burst is performed using the component stream cell counts and ACTT values. The details of this behaviour are explained for each simulation object starting in Section 3.4.

3.3.2 Basic rules for bursts

With a burst defined to be a cell stream describing cells each with the same averaged cell transmission time, some basic rules are required when dealing with `START` and `FINISH` messages in simulation objects. This is especially important as a burst may describe a number of cells less than or equal to the number reported in the `START` message. This will happen in the multiplexer, for example, as when a new cell stream arrives at an input, the current output burst has to have a `FINISH` message generated before a new `START` message, which includes the new input stream, is created. The number of cells in the burst which had the `FINISH`

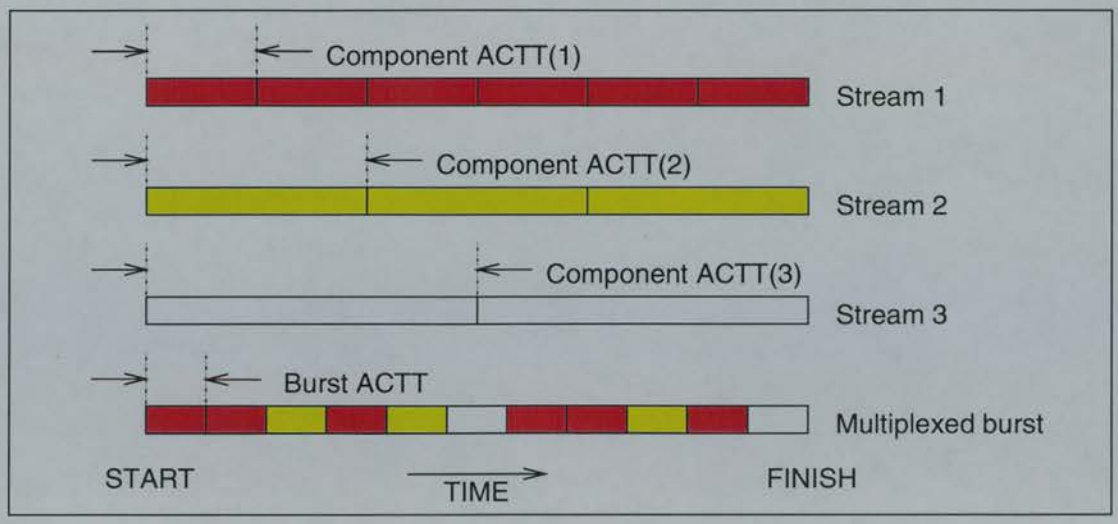


Figure 3.8: How a single burst represents more than one concurrent cell stream

message generated, may well be less than the number described in the previous START message. Thus, the full details of a burst are only known once a FINISH message arrives, making it important that objects can deal reliably with bursts when their knowledge is based solely on the START message. To aid this, the following basic rules, or assumptions, have been set for bursts:

A burst requires both a START and a FINISH message: the START message has the job of describing the maximum number of cells for a burst (hence the maximum cell count fields in the START message), while the FINISH message confirms what was actually transmitted.

Only one burst may by simulated on one communication link at any one time: overlapping START and FINISH messages for different bursts are not allowed on the same communication link. The burst multiplexer accepts overlapping START and FINISH messages, but the overlap is on *different* input links prior to the multiplexer.

A burst FINISH message may only have a simulation time greater than or equal to the START message for the burst: strict ordering of burst messages must be maintained to simplify the handling of events by simulation objects.

The total number of cells reported in a START message must be the maximum number of cells the burst may carry: a burst may not report more cells in the FINISH message than it does in the START message.

The time period of a burst may not be less than the total number of cells carried multiplied by the burst ACTT: the burst must carry a total number of cells less than or equal to the number described by the `START` message. Some flexibility is possible with the time period being slightly greater than the product of the total cell count and the burst ACTT. This is due to the simulation object trusting the data in the `START` message, making the `FINISH` essential if and only if the number of cells described by the burst is less than the maximum reported in the `START` . If the `FINISH` message arrives slightly "late", an object will not add extra cells to the burst as the maximum is number is set by the `START` message.

The total of the number of cells for each stream carried in a `START` message must equal the total for the burst: the sum of the cell contributions from each component stream must equal the total number reported.

The individual and component cell counts carried in a `FINISH` message must not be greater than those in the `START` message: the final number of cells sent by each stream must be less than or equal to the number originally reported in the `START` message.

With the preceding burst assumptions in mind, the simulation objects can be designed and implemented, such that their operation is as well defined as possible. This will aid the robustness of the simulator when performing experiments.

The following sections describe the design and implementation of the basic simulation objects necessary for performing burst-level simulation. Each object is implemented as a derived class of the generic class `object`, which provides an interface for scheduling events for other objects via the global event list. Other services, such as multi-level file logging of debugging information, are also provided by the `object` base class.

3.4 Multiplexing bursts

3.4.1 Overview

The technique used to multiplex bursts in a burst-level simulator is perhaps the most crucial when judging the accuracy of the technique. The basic aim is to collate coincident bursts on a shared communication link, so that the detail of each individual burst is not lost, and allowing for each to be recovered when required. The technique used in the simulator is to *split* input bursts into fragments (each

fragment also being a burst) and grouping coincident fragments as one burst at the output. Splitting input bursts produces fragments that are sequential in time, such that the simulation time span covered by the fragments is equivalent to the original burst. The multiplexer has to split input bursts to produce output burst messages when an input burst is either added to or removed from the output link (ie. upon the arrival of a `START` or `FINISH` at an input port). The individual bursts described in the current output burst of a multiplexer fully describe the concurrent bursts being received at the multiplexer input ports at that time.

Real multiplexers usually have a finite buffer capacity. This has been modelled by making the burst multiplexer a composite object which comprises a *burst concentrator* and an output *queue*. The burst concentrator has the job of producing output bursts which describe the currently active bursts at its input ports. The aggregate bandwidth of the multiplexed burst may well exceed the output link bandwidth of the multiplexer. The output queue has the job of policing the output bandwidth and performing cell loss, as appropriate, on the multiplexed output bursts. The implementation of the queue object is discussed in Section 3.6. Figure 3.9 shows the composite multiplexer object.

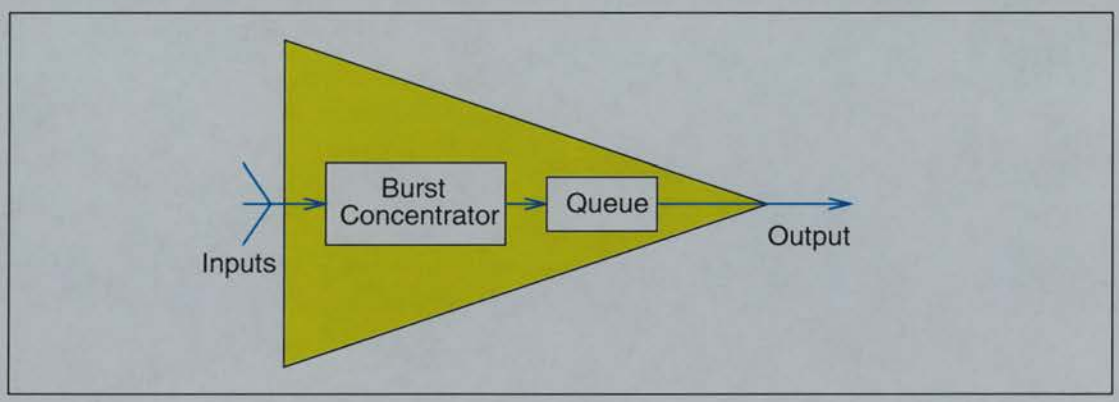


Figure 3.9: The composite multiplexer object comprising a burst concentrator and a queue object

The following two sections describe the techniques necessary for producing the output burst ACTT, and for splitting bursts into fragments. Techniques which exclusively use integer arithmetic have been developed for these purposes.

3.4.2 Calculating the burst ACTT

The burst concentrator in the multiplexer uses the ACTT value of each active input burst to determine the ACTT of the output burst to be produced. The *least common multiple* of all the coincident input bursts is calculated and is defined as

the *minimum sequence time* (*mst*). The *mst* can be calculated with Equation 3.1, where N is the number of coincident input bursts, where burst i has an ACTT value of $ACTT_i$.

$$mst = lcm(ACTT_1, \ldots, ACTT_N) \tag{3.1}$$

The least common multiple can be found for two numbers by using Equation 3.2, where $\gcd(x, y)$ is the greatest common divisor of x and y.

$$lcm(x, y) = \frac{x}{\gcd(x, y)} \times y \tag{3.2}$$

The minimum sequence time represents a time period in which each of the input bursts would transmit a whole number of cells spanning the entire period. As each burst will be sharing the communication link, the total number of multiplexed cells to be transmitted in this time period (C_T) can be determined by summing the cell counts for each input burst over the period. If each input burst i has an ACTT of $ACTT_i$, C_T can be calculated as follows:

$$C_T = \sum_{i=1}^{N} \frac{mst}{ACTT_i} \tag{3.3}$$

The ACTT of each cell in the multiplexed output burst ($ACTT_B$) is calculated to be the integer ceiling of the minimum sequence time, divided by the number of cells to transmit in this time. The value of $ACTT_B$ can be calculated with Equation 3.4.

$$ACTT_B = \left\lceil \frac{mst}{C_T} \right\rceil \tag{3.4}$$

As this technique is all integer, fractional ACTT values are not permitted, hence the reason for the integer ceiling on the final value of $ACTT_B$. This produces an over-estimate of the real ACTT, but the percentage error incurred can be reduced by the choice of the resolution of the integer clock. The greater the absolute size of the individual ACTT values used in this calculation (ie. the greater the resolution of the clock) the more accurate the value of $ACTT_B$.

Once the $ACTT_B$ has been calculated, this value is used to describe the multiplexed burst from that point onwards. Clearly, any input burst to the multiplexer may well contain less cells than the number required by Equation 3.3 to fill the calculated minimum sequence time. The result is that the output burst will be shorter than the minimum sequence time, but the duration is equal to the actual number of cells transmitted multiplied by $ACTT_B$. As $ACTT_B$ is an overestimate, this may well cause delay as the `FINISH` message for the burst has to be

scheduled to represent the entire duration of the burst. Any delay will cause the next `START` message to be delayed for scheduling until the preceding `FINISH` has been sent.

There is a potential problem with this technique as described, relating to the magnitude of the minimum sequence time. In a worst case situation, the individual ACTT values considered will have no common factors (apart from 1), making the minimum sequence time the *product* of all the component burst ACTT values (as shown in Equation 3.5). This is because if the greatest common divisor of two numbers is 1, the result of Equation 3.2 is the product of x and y.

$$mst = \prod_{i=1}^{N} ACTT_i \tag{3.5}$$

If the minimum sequence time is the product of the component burst ACTT values, integer overflow may occur within a few multiplications. Choosing a large integer variable to hold the minimum sequence time may help (eg. unsigned 64-bit) but this may still be a limitation. The maximal number of ACTT multiplications (M) which can be safely performed within integer precision is dependent on the size of the variable storing the mst (int_{MAX}) and the maximum size of an ACTT value ($ACTT_{MAX}$). M can be found with Equation 3.6.

$$M = \left\lfloor \frac{\log int_{MAX}}{\log ACTT_{MAX}} \right\rfloor \tag{3.6}$$

If, for example, the variable type used for the minimum sequence time is 64-bit (ie. `unsigned long long int`) and the maximum size for an ACTT is 1000, then M is as low as 6 as shown below:

$$M = \left\lfloor \frac{\log 2^{64}}{\log 10^3} \right\rfloor = 6$$

Multiplexing such a small number of input bursts, while preventing potential arithmetic overflow, could be a serious problem. A technique to entirely circumvent this limitation is shown in Section 4.5.2 on page 110.

3.4.3 Splitting bursts into fragments

Splitting bursts into fragments is also implemented as an integer technique. Burst splitting is used to break a burst into two time sequential fragments, each of which is also a burst. Each fragment contains a proportion of the cells from the original burst, with the proportion determined by the split time relative to the duration of the original burst. Each fragment contains an integer number of cells, and the

sum of the cell totals of each fragment equals the cell total for the original burst. A split is requested as a time period which is to be filled with cells from the original burst. Cells are allocated to the split time period such that the final duration of the fragment is at least equal to the split time duration (ie. the product of the number of cells in the burst and the burst ACTT is greater than or equal to the requested split time). The burst fragment produced is then a separate burst which can be passed to other objects in the simulator as required. The remaining burst fragment can be split again as it is also an independent burst.

A split time is rounded to the next cell boundary (ie. the nearest cell boundary at a time greater than the split time) as otherwise a `FINISH` message cannot be generated for a time greater than or equal to the split time. Splits can be performed on bursts where only the `START` message has been received, as all the information required is carried by the `START` . This allows objects, such as queues, to produce output bursts based on input bursts where only the `START` message has been received.

If a burst contains a single stream of cells, as shown in Figure 3.10, the decision on the number of cells to place in the split fragment is the obvious time based fraction of the total number of cells in the burst. The number of cells (C_S) returned for a split time period of T_{spl} is given by Equation 3.7. The burst `START` time is $T_{\texttt{STA}}$, the burst `FINISH` time is $T_{\texttt{FIN}}$, the burst contains C cells and the ACTT is $ACTT$. The integer ceiling is taken to round the split time to the next cell boundary.

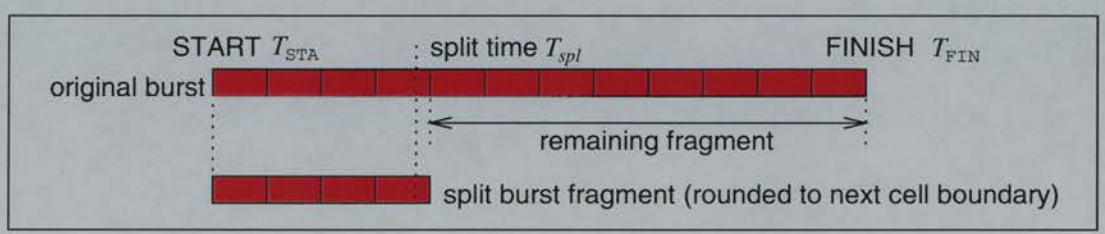


Figure 3.10: Splitting a burst containing one cell stream

$$C_S = \left\lceil \frac{T_{spl} - T_{\texttt{STA}}}{T_{\texttt{FIN}} - T_{\texttt{STA}}} \times C \right\rceil \tag{3.7}$$

However, as

$$T_{\texttt{FIN}} - T_{\texttt{STA}} = ACTT \times C$$

Equation 3.7 simplifies to Equation 3.8

$$C_S = \left\lceil \frac{T_{spl} - T_{\texttt{STA}}}{ACTT} \right\rceil \tag{3.8}$$

If the burst to be split contains more than one component stream (ie. a multiplexed burst), each component stream contributes to the total number of cells in the split fragment produced. Figure 3.11 shows a burst containing three component streams being split at an arbitrary point. The contribution of each component burst is based on the number of cells that component contributes to the multiplexed burst (ie. the more cells the burst contributes, the greater its contribution to the fragment). In terms of the ACTT values of each component, the lower the value, the greater the number of cells in the component.

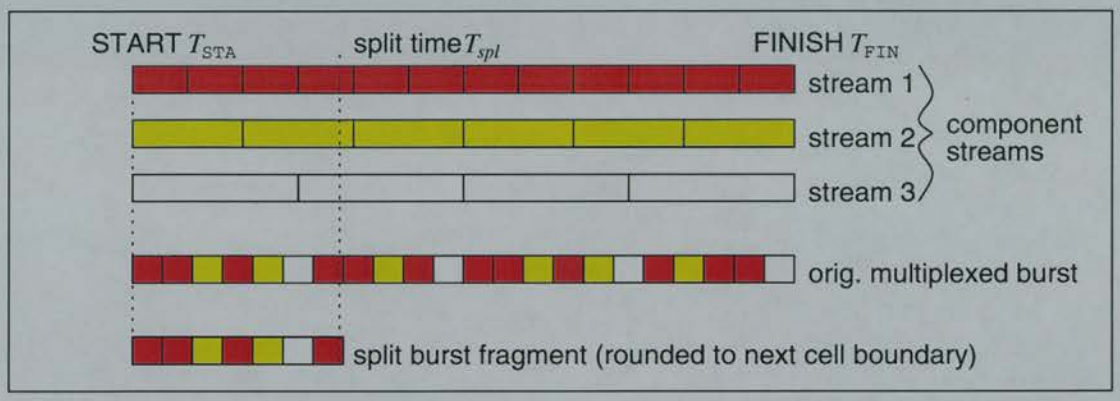


Figure 3.11: Splitting a burst containing more than one cell stream

There are several options for determining the fine detail of the split, but the technique chosen allocates fairly based on the number of cells a burst will have transmitted at the time of the split. When a multiplexed burst is to be split into fragments, the split time is aligned to the next cell transmission boundary for the burst (as in the case of a single stream burst). In other words, a fragment can only contain an integer number of cells such that the duration of the fragment is an integer multiple of the burst ACTT. The number of cells to transmit in the fragment (C_S) is calculated with Equation 3.9, where $ACTT_B$ is the burst ACTT.

$$C_S = \left\lceil \frac{T_{spl} - T_{\mathrm{STA}}}{ACTT_B} \right\rceil \tag{3.9}$$

When splitting a burst with more than one component cell stream, it is important to round the split time up to the next cell boundary greater than the split time. This is because the contribution of cells from each of the component bursts to the fragment is dependent on this time. The rounded split time, T'_{spl}, can be determined by using Equation 3.10.

$$T'_{spl} = T_{\mathrm{STA}} + (C_S \times ACTT_B) \tag{3.10}$$

Each component stream in the burst contributes a time based fraction of cells (in the same way as shown in Equation 3.8) in order to produce a fragment with C_S cells. The first contribution from stream i, c_i, is calculated as the integer floor of the split time period divided by the ACTT for the stream ($ACTT_i$) as shown in Equation 3.11. The sum of the first contributions, C_{first}, for N component streams is calculated with Equation 3.12.

$$c_i = \left\lfloor \frac{T'_{spl} - T_{\text{STA}}}{ACTT_i} \right\rfloor \tag{3.11}$$

$$C_{first} = \sum_{i=1}^{N} c_i \tag{3.12}$$

The total number of cells from the first contribution might not equal the number of cells required for the fragment (ie. $C_{first} < C_S$). The shortfall is made up by including cells from streams which have in addition contributed the largest "fraction" of a cell at the split time. As this is an all-integer technique, the notion of a cell fraction has to be clearly defined. The approach taken has been to reduce the cell fraction calculation for each component stream to one which has a common denominator across all the component streams. Choosing the component cell stream with the largest fraction of a cell is then a matter of choosing the stream with the largest fraction numerator. The approach is detailed below.

The fraction of a cell, f_i, contributed by each burst stream is the time difference between the split time and the time spanned by the first contribution of cells (ie. c_i multiplied by $ACTT_i$) divided by the ACTT for the stream $ACTT_i$. Equation 3.13 can be used to find f_i.

$$f_i = \frac{(T'_{spl} - T_{\text{STA}}) - (c_i \times ACTT_i)}{ACTT_i} \tag{3.13}$$

To produce a common denominator for all of the fraction equations, the top and bottom of Equation 3.13 are multiplied by $(mst/ACTT_i)$ giving Equation 3.14. The minimum sequence time (mst) is a constant which is larger than any component ACTT. In some cases using the mst may not be practical, so an alternative is to multiply the top and bottom of Equation 3.13 by $\lfloor ACTT_M/ACTT_i \rfloor$ instead, where $ACTT_M$ is the largest component ACTT value present in the multiplexed burst. This ensures that f_i can always be represented by an integer greater than zero for every component stream with a cell fraction available.

$$f_i = \frac{[(T'_{spl} - T_{\mathtt{STA}}) - (c_i \times ACTT_i)] \times \frac{mst}{ACTT_i}}{mst} \tag{3.14}$$

By throwing away the denominator (be it mst or$ACTT_M$) and just using the numerator, relative cell fractions for each component burst can be calculated and compared. The desired cell total of C_S for the fragment is reached by repeatedly incrementing by one the c_i value with the greatest fraction. Once a stream has been incremented due to its fraction, the value of f_i for that stream is set to zero to ensure that it is not considered again. This ensures that cells are allocated to streams in an order with the largest fraction first and the least fraction last (if that many additions need to be made).

The example split in Figure 3.11 shows the technique in action. Firstly, the split time is rounded to the next cell boundary for the fragment produced. Secondly, cell fractions have to be considered when determining the relative contributions of each component stream. The first allocation of cells provides 5 cells (3 from stream 1, and one each from streams 2 and 3) and the large cell fractions contributed by streams 1 and 2 are used to provide the final 2 cells for the fragment. Once again, the ordering of cells shown in the multiplexed burst and the split fragment are for illustration purposes only (as no relative ordering information is carried in a burst, only individual stream cell counts).

One consequence of burst splitting, is that it may produce burst fragments which describe zero cells in one or more of the component streams. In the worst case, all of the cell counts are zero but this usually implies that the split time period was zero. A more likely scenario is when a small split time period is requested, such that cells are allocated from cell streams with small ACTT values, and not from streams with relatively larger ACTT values. The capability of dealing with zero cell length bursts needs to be borne in mind when designing simulation objects which may encounter them.

3.4.4 Burst concentrator behaviour

With the techniques defined for calculating the burst ACTT for a group of concurrent bursts, and for splitting bursts into fragments, the behaviour of the burst concentrator object can be defined. In essence, a change of state at the burst concentrator output port is triggered by a change of state at one of its input ports. An input port has two states; dealing with an incoming burst or being idle. The output burst also has two states; transmitting a burst or being idle. An input port stops being idle when a `START` message arrives, and becomes idle when a `FINISH` message arrives. The output port stops being idle when a `START` message

is produced, and becomes idle when a FINISH message is produced. Output burst production is governed by the arrival of START and FINISH messages at the input ports.

The following actions are taken when a START message arrives at an input port:

1. An appropriate *input* object (see Section 3.4.5) for the port is updated to reflect the status of the new input port. An estimated finish time is calculated for the burst.

2. The output port of the burst concentrator is checked to see if a burst is being transmitted. If so, the current output burst has a FINISH message generated for it.

3. The new input burst is added to the output burst. This involves recalculating the ACTT for the output burst and finding the cell counts for the maximum possible output burst.

4. A START message is generated for the output port for the new output burst.

When a FINISH message arrives at an input port, the following actions are taken:

1. The status of the *input* object for the port is updated to reflect the actual cell stream counts for the completed burst.

2. A FINISH is generated for the current output burst to reflect that fact that one of its component input bursts has finished. Cell counts are calculated to fill the duration of the output burst.

3. The input object for the input port is changed to signify that there is no longer a current input burst.

4. If the output burst still has active input streams (from other input ports) a new START message is generated at the output. This involves calculating the new burst ACTT and finding the cell counts for the maximum possible output burst.

As previously described, the burst ACTT value is potentially an overestimate of the optimal value (due to taking the integer ceiling). An effect of this approximation is that the production of START and FINISH messages at the burst concentrator output port will be delayed relative to the timings of arriving input messages. Account of any relative delay is taken when producing output

burst messages such that the strict time ordering of `START` and `FINISH` messages is obeyed. Cell contributions from each input burst are taken relative to the timings of each input burst (ie. by using the "last contribution time"), so message time management at the output port does not affect the cell contribution calculations.

3.4.5 The design of the burst concentrator

The "internals" of the burst concentrator are shown in Figure 3.12. The job of the burst concentrator is to merge cell streams arriving at the input ports, to produce bursts at the output port. As shown in Figure 3.12, the burst concentrator only has one input port and one output port (ie. an object scheduling an event for the multiplexer is given one pointer to the multiplexer input port). Determining the input port for an arriving `START` or `FINISH` message, is done by examining a "from" variable in each new message, which is set by the original scheduling object. Simulation objects which provide input for the multiplexer are assigned a suitable "from" index (to map to a virtual multiplexer input port) when they are instantiated. This enables the multiplexer to keep track of which input ports are active when it only has one "real" input port.

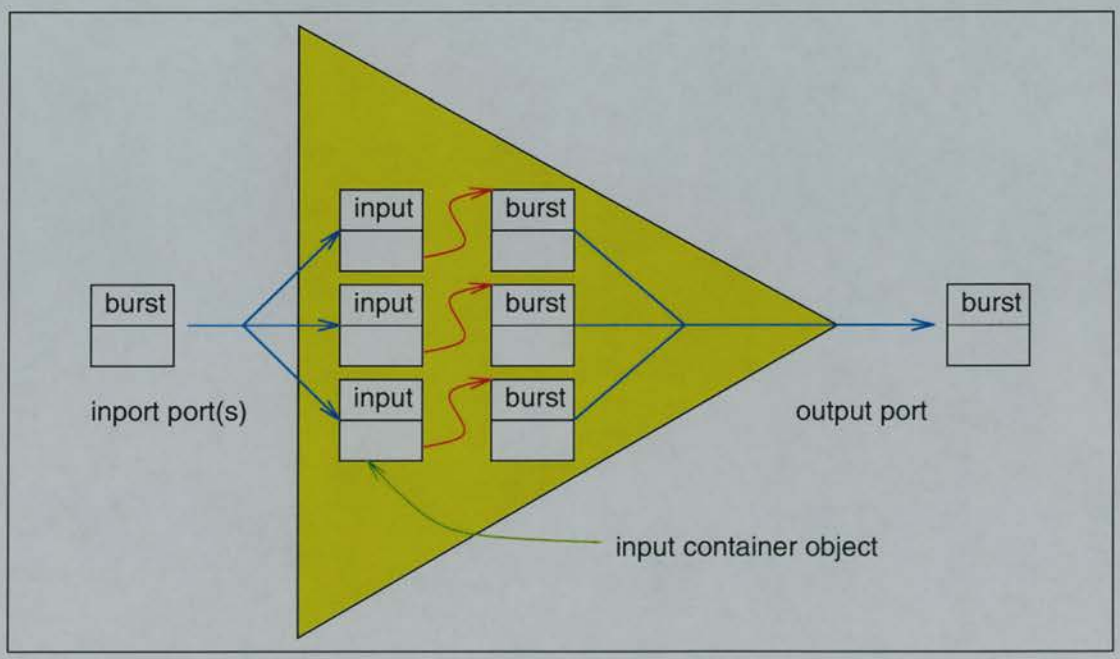


Figure 3.12: The logical structure of the burst concentrator part of a multiplexer simulation object

As an input burst for a multiplexer may contain one or more cell streams, the burst concentrator uses the burst ACTT and total cell count values when deter-

mining the contribution of each input burst to the current multiplexed output burst. Burst ACTT and total cell count values are common features of all bursts regardless of the number of cell streams they describe. The burst ACTT and total cell count are a "meta" representation of a burst, effectively describing the one or more component cell streams as one cell stream. This allows for double cell splitting to be used, with a split first performed on the "meta" representation, and then using the result to split the individual cell streams contained in the burst. The result is a split burst fragment which contains the correct proportions of the component cell streams in the burst for the split time period. For a single burst, secondary cell splitting is unnecessary but this technique is useful when splitting input bursts for multiplexed output bursts. The "meta" input bursts are split as necessary to fill the output burst and once the counts are assigned, burst splitting is used to allocate this contribution among the component streams of the input burst. The final output burst contains all of the individual cell streams described by the concurrent input bursts, with the correct cell proportions for the time period of the output burst. In essence, the calculation for the make-up of the output burst is a two-tier process:

- Firstly, the "meta" description of each input burst is used to determine how many cells each input burst contributes to the output burst. Burst splitting is used, including adding fractions of "meta" input cells to input burst allocations.

- Secondly, the cell contribution calculated for each input burst is distributed among the individual cell streams carried by that input burst. Burst splitting is also used here.

To keep track of the use of the "meta" input burst and the cell streams each one describes, the burst concentrator maintains an *input* object for each active input port. The input object holds a pointer to the burst `START` message associated with the current burst at that input. When the burst concentrator calculates the make-up of an output burst, the input object for each port is interrogated for the burst ACTT and total cell count information which are then used to assign a cell count to each input burst. When this job has been performed, the count assigned to each input is used to get cell stream contributions from the burst object pointed to by the input object. Once an output burst has been produced, each input object performs the required housekeeping to ensure that it accurately represents the current status of its input burst. This includes logging a "last contribution time" which is the cell boundary for the first cell which

has not been transmitted at the output yet. Using this time as one endpoint, when considering cell contribution time periods, gives bias to bursts whose cell fraction contributions were not used in the previous output burst. This helps ensure that input bursts with relatively small ACTT values are not given unfair priority by burst splitting. When a `START` message arrives at an input port, the "last contribution time" is set to that time for that input. When an output burst `FINISH` message is created, the "last contribution time" for each input burst is incremented by the product of the burst ACTT and the total cell contribution for that burst.

As asserted in Section 3.3.2, a `START` message has to state the maximum number of cells the burst may possibly transmit. The multiplexer has to obey this, so production of an output `START` message requires an extra step. In addition to finding the output burst ACTT, the input burst which is estimated to finish *earliest* is found, and this time period is used for estimating the *maximum size* of the output burst. When a new input burst is registered with its input port object, the estimated finish time, T_{est} can be calculated with Equation 3.15 (where $T_{\texttt{STA}}$ is the `START` message time, $ACTT_B$ is the burst ACTT and C_T is the maximum (total) number of cells in the burst (as reported in the `START` message)).

$$T_{est} = T_{\texttt{STA}} + (C_T \times ACTT_B) \tag{3.15}$$

Once the smallest estimated input burst finish time is found, the difference between it, and the current time, is the maximum time duration for the output burst. Of course, any input may a receive a `FINISH` message before the earliest estimated finish time, but this still means the output `START` message describes the maximum burst size possible. The two-tier burst splitting technique is used to fill in the cell stream counts for the output burst `START` message from the current input bursts. To find the maximum number of cells each input burst may contribute to the output burst, burst splitting of the "meta" input burst is performed with cell fractions used if necessary. Once a set of cell counts has been found which fills the required time period, burst splitting is used again to produce the fine detail of the cell contribution from each cell stream in each input.

When the burst concentrator has to produce a `FINISH` message, the time period used to calculate each input burst's cell contribution is the finish time for the burst minus the "last contribution time" for the burst. The "last contribution time" is calculated as the arrival time for the burst plus the time duration of the cells which have already been sent from that burst (ie. the original ACTT for the burst multiplied by the number of cells which have been sent from the

burst). This is used in preference to the output burst `START` message time, as bias is given to input bursts which may have been overlooked when cell fractions were added to the previous output burst. The bias is in the form of a slightly longer time period, which is used to determine the cell contribution of the burst to the current output burst. Bursts which previously added cells by fractions have a correspondingly shorter time period, which is used to calculate their cell contribution to the current output burst. The two-tier burst splitting is then used to calculate the number of cells contributed from each cell stream in each input burst.

3.5 Demultiplexing bursts

3.5.1 Overview

The job of the demultiplexer is to receive sequences of bursts, each described by an individual pair of `START` and `FINISH` messages, and route the individual cell streams within the burst to its outputs. An input burst may contain one or more individual cell streams, each of which may have a separate demultiplexer output port destination. More than one individual cell stream from an input burst may have the same destination output port. To deal with this, the demultiplexer needs limited burst multiplexing capability, although the operations required are not as complex as for the burst multiplexer described in the previous section. The bursts produced at each demultiplexer output port are all fully fledged bursts described by an individual `START` and `FINISH` message pair.

A real multiplexer may have different bandwidths and buffering on each output port. To simulate this, a compound simulation object, as shown in Figure 3.13, may be used. The demultiplexer produces bursts for each output port (depending on the make-up of the input burst) and the individual queue objects are used to police the bandwidth, performing cell loss calculations as appropriate.

The following section describes the design and implementation of the demultiplexer object in the simulator.

3.5.2 The demultiplexer design and behaviour

The logical structure of the burst demultiplexer is shown in Figure 3.14. The demultiplexer accepts bursts at its input and produces individual bursts at each of its outputs. Unlike the "virtual" input ports of the multiplexer, the demultiplexer has a unique port for each output port modelled. Other simulation objects are attached to these ports (ie. burst events scheduled at an output port will

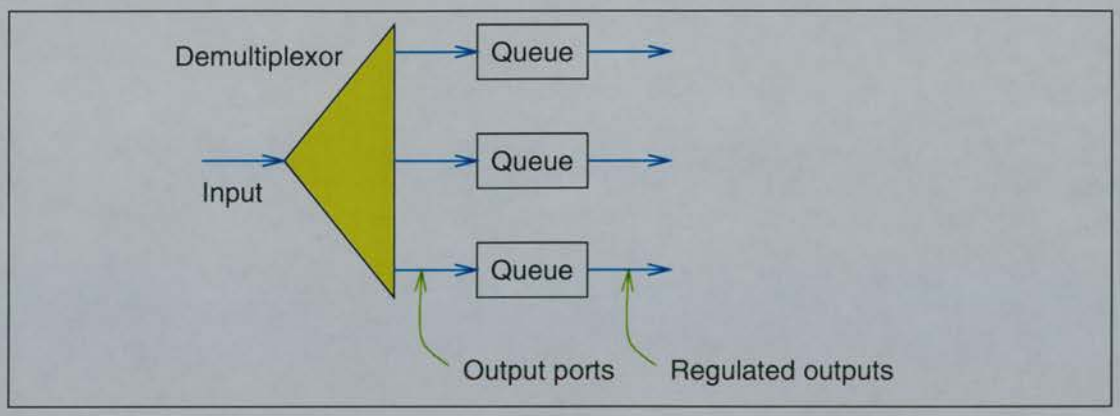


Figure 3.13: The compound demultiplexer object where each output port is regulated by a queue object

have a unique simulation object as their destination). If bandwidth limiting of each output port is required, a compound demultiplexer with queues attached to each output port is used (as shown in Figure 3.13). Each queue has the job of limiting the bandwidth of bursts delivered from each demultiplexer output port, performing any cell loss as appropriate. Other simulation objects, for which the policed bursts are destined, are attached to the output port of each queue.

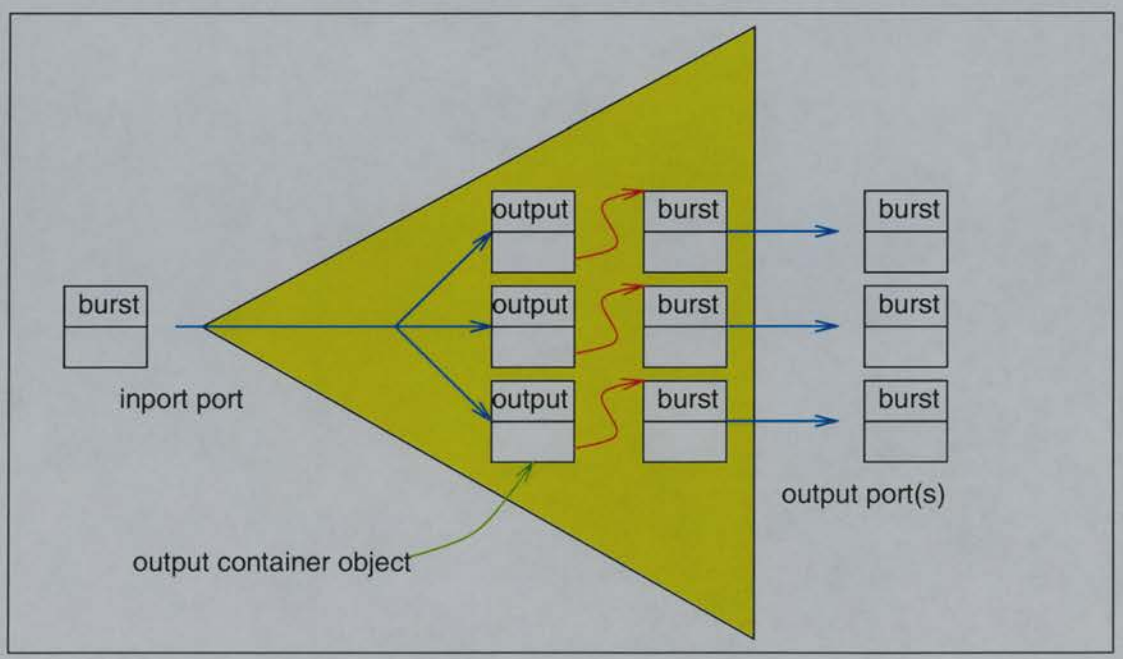


Figure 3.14: The logical structure of the demultiplexer

A demultiplexer is effectively a burst *switch*, as each individual cell stream in an input burst may be mapped to any demultiplexer output port. To appropriately manage this, the demultiplexer uses an "output" class to control the current

burst on each output. When an input burst arrives (ie. a `START` message), the individual cell streams are checked in turn against an output *port mapping table* and the details of each stream are passed to the output object for the relevant port. When the entire contents of the input `START` message have been processed, each output object produces and schedules an appropriate `START` message for its output port. The only calculation performed is that of finding the ACTT for the cell streams multiplexed on each output port. This involves using the technique described in Section 3.4.2 on the cell stream ACTT values for each output object. Other information, such as the maximum number of cells each stream may carry, is taken directly from the information in the input burst `START` as this will not change. This is because the demultiplexer only receives a single stream of bursts at its input, and thus can apply the basic burst assumptions (as per Section 3.3.2).

When the `FINISH` message arrives for the current input burst, the final cell count details of each stream stored in the output objects are suitably updated. Each output object then creates and schedules a `FINISH` message for its output burst. The goal is to schedule the `FINISH` message for each output port, so that the duration of each burst matches the product of the cell count and the burst ACTT. However, each output burst produced may be shorter or longer than the input burst from which it is derived. This is due to the overestimate in the original input burst ACTT, and the addition of cells based on fractions to the input burst. When the cell streams from an input burst are split by the demultiplexer, the ACTT for each output burst is recalculated if there is more than one component cell stream in the output burst. When this happens, the duration of each output burst may not match the duration of the input burst from which each is derived. This is because each new ACTT calculated is also an approximation, and the original input burst is not a perfect representation of the cell streams it describes. If an output burst describes a single cell stream, the same problems apply.

An output burst will be *longer* in duration than the input burst if the product of its ACTT and the total cells carried is larger than the duration of the input burst. This will cause delay in the scheduling of `START` and `FINISH` messages at that output port. An output burst will be *shorter* in duration than the input burst, if the product of its ACTT and cells carried is less than the duration of the input burst. In this case, the output `FINISH` message, generated when the input `FINISH` message arrives, will be scheduled for a simulation time *after* the correct time (ie. the time of the `START` message for the output burst plus the product of the output burst ACTT and the number of cells carried). Messages

cannot be scheduled at a time less than the current simulation time which will be the arrival time of the input FINISH message in this case. Such a burst will be called an *overlong* burst in this work, and overlong bursts are discussed further in Chapter 5.

The absolute difference in duration of each output burst to the input burst will typically be less than one output burst ACTT, depending on the accuracy of the original input burst ACTT. The higher the accuracy, the better the burst will be as a representation of the multiplexed cell streams it describes. This leaves the greatest factor influencing the error in duration of each output burst to be cells added by fractions. With this in mind, the problems of overlong bursts can be addressed through the use of output delays or output queues (as in Figure 3.13). Any delay introduced to model transmission line latency or the passage of cells through a queue will help reduce the absolute difference between the time of the FINISH message and its ideal time (as defined to be the START time for the burst plus the product of the total cells carried and the burst ACTT).

The demultiplexer provides an ideal opportunity to remove cell streams which describe a zero number of cells. This is relatively straightforward when the maximum cell count for a stream is reported as zero in the input START message. If a demultiplexer output is presented with a single zero cell stream, then no output START message need be generated. When a single demultiplexer output describes more than one cell stream, any streams containing zero cells can be optionally excluded from the outgoing burst. If the input FINISH message describes cell streams with zero cells, then it is a different matter. If the preceding input START message contained a non-zero cell count for the stream eventually containing zero cells, a FINISH message must be generated at the output. A single link can only contain sequences of START and FINISH message pairs and this must be maintained even if the burst contains zero cells. Any zero cell streams which are removed by the demultiplexer help reduce the number of bursts which must be simulated. The fewer bursts which need to be simulated, the less work the simulator will have to do. Zero cell length bursts can also be removed if an output delay, or output queue, is used on each output port. If a FINISH message describing zero cells arrives at a simulation time *before* the scheduled time of the START message (ie. due to a parameterised delay), the START message can be cancelled.

3.6 Queueing bursts

3.6.1 Overview

A flexible technique to model the operation of queues and buffers is essential in a communication network simulator. A generic queue object can be used to provide buffering in other simulation objects. For example, the compound multiplexer as described in Section 3.4 requires a burst concentrator to produce the compound bursts, and a queue object to model the finite buffer space in the multiplexer and a peak output bandwidth. The queue is a single simulation object which may be used independently or as part of a compound object. Figure 3.15 shows the basic queue.

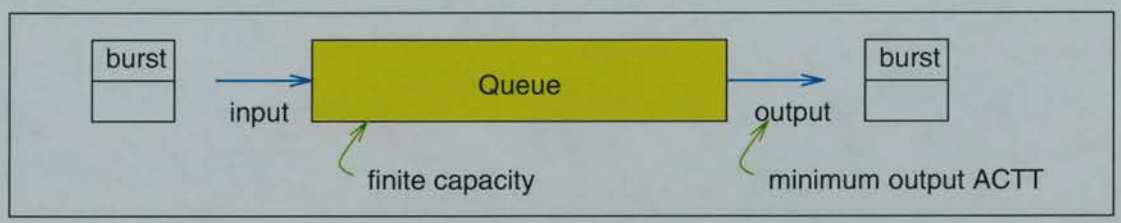


Figure 3.15: The basic queue simulation object

Several features of the integer-time burst-level simulation approach make the modelling of queues relatively straightforward. Every object, apart from the multiplexer burst concentrator, sees arriving burst streams strictly as sequential `START` and `FINISH` pairs. As the queue also has to produce output `START` and `FINISH` messages in strict order, decisions on effects such as cell loss from bursts traversing the queue can be delayed until all of the characteristics of the incoming burst are known (ie. when the input `FINISH` message arrives). This means that such calculations can be exact, as they are based on the precise details of the input bursts.

Like the demultiplexer, the queue object has the potential for eliminating zero cell length bursts from a communication link. The ability to do this is dependent on the configuration choices made for the queue. Ideally, the queue will introduce some parameterised *delay* to each burst passing through it. The delay can be thought of as some physical propagation delay for cells traversing the queue (for example). In this case, the queue will receive and forward incoming burst `START` message as per normal, but with a delay added to the scheduled time of the output `START` message. If the input burst `FINISH` message arrives at a time *less* than the scheduled time of the output `START` message, the output `START` can be removed if the input burst describes zero cells. This action will save other simulation objects

connected to the queue output port from having to waste time dealing with zero cell length bursts. Zero bursts can also be removed if previous bursts are still in the queue when the new burst arrives. In this case, the `START` message for the new burst can only be transmitted at a time greater than or equal to the the `FINISH` message for the previous burst. If the input `FINISH` message arrives before this time, and the input burst describes zero cells, then the burst can be removed from the simulation.

Two varieties of queue have been implemented in the simulator, both of which follow the same idea. The first type of queue performs no dialogue with its consumer, whereas the second type negotiates an output ACTT with the consumer before starting to send bursts to it.

3.6.2 Scaling component ACTT values in queues

In the simulator design, bursts are described by a single burst ACTT value and a maximum number of cells, regardless of the number of cell streams contained within. This makes the operation of other simulation objects simple as they initially use these few parameters. However, when finely detailed behaviour is modelled (eg. when splitting a burst), the individual stream information needs to be used. The queue object also works on the "meta" burst level, but needs to apply any changes it makes to the whole burst to the individual streams contained within. Cell loss is performed by splitting a burst into two fragments, one of which represents the lost cells. The burst splitting procedure then allocates the cell loss over the component cell streams carried in the burst. The "lost" burst fragment is discarded, while the remainder of the burst, representing the cells which were not lost, is then passed on to subsequent simulation objects. The burst splitting technique used is as described in Section 3.4.3. The other effect the queue may have on a burst is to *alter* the burst ACTT. This may occur when the burst queues, and when the queue output ACTT is greater than that of the burst. As the burst ACTT value is calculated from the individual cell streams contained in the burst, some means of altering each individual cell stream ACTT value, to reflect a change in the burst ACTT, is required. One such means is outlined as follows.

If an input burst to the queue contains C_T cells, has a burst ACTT of $ACTT_B$ and the new output burst ACTT for the burst is $ACTT'_B$, then the ACTT for component cell stream i ($ACTT_i$) is given by Equation 3.16, where c_i is the number of cells contributed by that component.

$$ACTT_i = \frac{C_T \times ACTT_B}{c_i} \Rightarrow c_i = \frac{C_T \times ACTT_B}{ACTT_i} \tag{3.16}$$

Equation 3.17 shows how $ACTT_i'$ (the new value for the component ACTT value scaled to take account of the change in the burst ACTT) can be found with a substitution made for c_i.

$$ACTT_i' = \frac{C_T \times ACTT_B'}{c_i} = \frac{C_T \times ACTT_B'}{\frac{C_T \times ACTT_B}{ACTT_i}} = \frac{ACTT_B' \times ACTT_i}{ACTT_B} \tag{3.17}$$

Equation 3.17 shows that individual component ACTT scaling can be simply based on the old and new values of the burst ACTT. No explicit cell counts are required, thus other objects in the simulator can perform appropriate burst component ACTT scaling solely on the information contained in a `START` message.

3.6.3 Queue without consumer flow control

The first type of queue to consider is the one in which the queue immediately starts to produce output bursts upon the receipt of bursts at its input port. No output request dialogue is performed with the consumer object for the queue. This is the simplest form of the queue, and the queueing operation can be narrowed down to dealing with four different scenarios as shown in Figure 3.16.

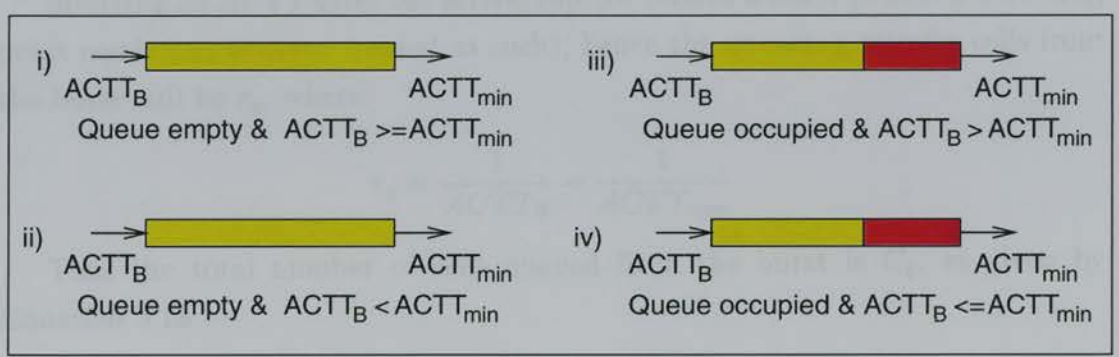


Figure 3.16: The four queue scenarios

For each queue scenario it is assumed that the ACTT of the burst entering the queue is $ACTT_B$, and the minimum burst ACTT at the queue output is $ACTT_{min}$. The minimum burst output ACTT is the shortest cell transmission time the queue can produce, and thus controls the maximum bandwidth of the output. Section 3.6.2 details how a change to the burst ACTT, due to queueing, can be applied to the individual stream ACTT values in a burst containing more than one cell stream.

An burst input to the queue is triggered by the arrival of a START message at the input port, and is terminated by the arrival of the associated FINISH message. The queue generates bursts by scheduling START and FINISH message pairs for the object connected to its output port.

Scenario i: queue empty & $ACTT_B >= ACTT_{min}$

In this case the queue is empty (ie. any previous bursts have completely exited the queue) and the ACTT of the input stream, $ACTT_B$, is greater than the queue output stream ACTT, $ACTT_{min}$). There will be no queueing, and the burst can be forwarded to the queue output with a burst ACTT equal to $ACTT_B$.

Scenario ii: queue empty & $ACTT_B < ACTT_{min}$

In this case the input burst has a lower ACTT than the output ACTT for the queue (ie. a faster cell input rate), so cells from the burst will accumulate in the queue. The output ACTT for the burst will be $ACTT_{min}$. Assuming there are C_T cells in the burst, then the total time for all of the cells to arrive at the queue will be T_B, where:

$$T_B = C_T \times ACTT_B$$

Inverting an ACTT gives the arrival rate for cells in a burst (which is a floating point result but is never treated as such), hence the queueing rate for cells from the burst will be r_q, where:

$$r_q = \frac{1}{ACTT_B} - \frac{1}{ACTT_{min}}$$

Thus the total number of cells queued from the burst is C_q, as given by Equation 3.18.

$$C_q = \left\lceil (C_T \times ACTT_B) \times \left(\frac{1}{ACTT_B} - \frac{1}{ACTT_{min}} \right) \right\rceil = C_T - \left\lfloor \frac{C_T \times ACTT_B}{ACTT_{min}} \right\rfloor \tag{3.18}$$

If the queue has a capacity of Q cells, then if $C_q > Q$ the cell loss from the burst is L, where:

$$L = C_q - Q$$

The *queue empty time* (QET) can be defined as the time at which the FINISH message for the queued burst will be transmitted. If the START message for the burst is $T_{\texttt{STA}}$, then the QET can be calculated with Equation 3.19.

$$QET = T_{\texttt{STA}} + [(C_T - L) \times ACTT_{min}] \tag{3.19}$$

The queue can wait for the input `FINISH` message to arrive before it has to perform the calculations above, as the receipt of the `FINISH` message confirms the C_T count for the burst. If the burst contains multiple component streams, burst splitting (as described in Section 3.4.3) is used to appropriately allocate the total loss over each component stream. The final action is to schedule the `FINISH` message to mark the end of the burst at the queue output port. The output `FINISH` message is scheduled for the QET which becomes the earliest first send time for a subsequent input burst.

Scenario iii: queue occupied & $ACTT_B > ACTT_{min}$

In this case the queue still contains cells from a previous burst (or bursts) and the ACTT for the new burst is greater than the queue output ACTT value. The result of this is that no cells from the new burst will be lost (as there necessarily must always be space for one new input cell), but the effect on the output ACTT of the new burst is more complicated than in the other three scenarios. As the queue already contains cells from previous bursts which are exiting the queue with the minimum ACTT allowed ($ACTT_{min}$), a proportion of the cells in the new burst will queue behind these previous cells, and thus will be transmitted at $ACTT_{min}$ when they reach the output. If emptying the queue of the previous burst cells takes longer than T_B (ie. the time to fully receive all of the cells from the new input burst), then all of the cells from the new input burst will have an output ACTT of $ACTT_{min}$. If emptying the queue of the previous burst takes less than T_B then some proportion of the cells from the new input burst will have an output ACTT of $ACTT_{min}$, whereas the remainder will have an output ACTT of $ACTT_B$.

If a previous burst leaves cells in the queue after receipt of the `FINISH` message for that burst, the queue updates a variable recording the QET for the burst. Hence, to calculate the number of cells, C_q, from a new burst (arriving at time $T_{\texttt{STA}}$) which will queue behind the previous burst, Equation 3.20 can be used.

$$C_q = \left\lceil \frac{QET - T_{\texttt{STA}}}{ACTT_B} \right\rceil \tag{3.20}$$

However, the estimate provided by Equation 3.20 is a poor approximation as it discounts the fact that more cells from the new burst can queue as the C_q cells are transmitted with an ACTT of $ACTT_{min}$. An iterative algorithm could be used to find the maximum C_q for the new input burst using Equation 3.21.

$$C_{q_i} = \frac{(QET - T_{\mathtt{STA}}) + (C_{q_{i-1}} \times ACTT_{min})}{ACTT_B} \tag{3.21}$$

However, Equation 3.21 is simply a geometric series (with $a = \frac{QET - T_{\mathtt{STA}}}{ACTT_B}$ and $r = \frac{ACTT_{min}}{ACTT_B}$) which can be summed (ie. $S_\infty = \frac{a}{1-r}$) to give the final result for C_q as calculated in Equation 3.22.

$$C_q = \left\lfloor \frac{QET - T_{\mathtt{STA}}}{ACTT_B - ACTT_{min}} \right\rfloor \tag{3.22}$$

If $C_q < C_T$ the burst splitting technique is used to produce two fragments to describe the cells transmitted at $ACTT_{min}$ and $ACTT_B$. The `START` for the $ACTT_{min}$ fragment is scheduled first at the QET. The `FINISH` message for the $ACTT_{min}$ fragment, as well as the `START` message for the $ACTT_B$ fragment are generated at T_{change} where:

$$T_{change} = QET + (C_q \times ACTT_{min})$$

The `FINISH` for the $ACTT_B$ fragment is generated at the new QET (which is calculated for the end of the $ACTT_B$ fragment) as follows:

$$QET = T_{change} + [(C_T - C_q) \times ACTT_B]$$

If, instead, $C_q >= C_T$ (ie. the entire burst is queued to be output with an ACTT of $ACTT_{min}$) then the QET has to be adjusted to reflect this and the appropriate `FINISH` message generated. If the previous QET was QET_{prev}, the new QET is:

$$QET = QET_{prev} + (C_T \times ACTT_{min})$$

As the `FINISH` message for the input burst may arrive early (cf. the information in the `START`), the queue needs to take account of this when sending the output burst fragments. If the input `FINISH` arrives before, or at, T_{change} (ie. the input burst cell count is less than or equal to the number of cells to have been sent in the $ACTT_{min}$ fragment), the `START` for the $ACTT_B$ fragment needs to be cancelled and the `FINISH` message for the $ACTT_{min}$ fragment may need to be rescheduled. It the input `FINISH` arrives after T_{change}, the `FINISH` message for the $ACTT_B$ fragment can be scheduled as appropriate.

Scenario iv: queue occupied & $ACTT_B <= ACTT_{min}$

In this case the queue contains cells from a previous burst (or bursts) exiting the queue with an ACTT of $ACTT_{min}$, and the new input burst has an ACTT

greater than the output queue ACTT ($ACTT_{min}$). Unlike the behaviour described in scenario iii, there is no need to split the new input burst to model a change in the output ACTT as $ACTT_B < ACTT_{min}$. This means that all of the output cells will have an ACTT of $ACTT_{min}$ for the burst.

The first step is to determine how many cells remain from the previous burst or bursts. The number remaining, C_{rem}, can be calculated as follows for a new input burst `START` message arriving at $T_{\texttt{STA}}$:

$$C_{rem} = \left\lceil \frac{QET - T_{\texttt{STA}}}{ACTT_{min}} \right\rceil$$

To calculate the number of cells which will queue from the new burst, Equation 3.18 can be used. Hence the total number of cells to queue, C_q, is:

$$C_q = C_{rem} + C_T - \left\lfloor \frac{C_T \times ACTTb}{ACTT_{min}} \right\rfloor$$

If $C_q > Q$ where Q is the capacity of the queue, then there will be cell loss from the new input burst. The cell loss, L, is calculated as:

$$L = C_q - Q$$

Hence the new QET for the queue, with a previous QET of QET_{prev}, can be calculated as:

$$QET = QET_{prev} + [(C_T - L) \times ACTT_{min}]$$

Once again, if the new burst contains more than one component stream, burst splitting is used to fairly allocate the loss over the component streams. A suitable `FINISH` message for the burst is scheduled for time QET at the queue output.

3.6.4 Queue with consumer flow control

The second class of queue is the one where the queue consumer object performs flow control negotiation with the queue before accepting bursts. To simplify the design of the queue, a very simple protocol is used for communication between the queue and its consumer. If the queue is empty, the arrival of an input `START` message causes the queue to send a request (`REQ`) message to the consumer. If the consumer wishes to accept an output burst, it replies with an `OK` message containing both a minimum ACTT value and a maximum number of cells for the output burst. The queue object then sends a burst (ie. a pair of `START` and `FINISH` messages) detailing a burst within the parameters specified by the consumer. If

the input burst is larger than the maximum number of cells requested by the consumer, the queue initiates the entire request sequence repeatedly until the input burst is sent in its entirely (excluding cells which have been lost). If more than one burst is in the queue, the final `FINISH` message for the burst at the head of the output is followed by a `REQ` for the next burst in the queue. This occurs even if the final burst fragment from the previous burst contained less cells than detailed in the last `OK` message from the consumer.

Figure 3.17 shows the internal structure of the queue with flow control. The flow controlled queue differs to the non-flow controlled version in that each input burst is controlled by a `qint` object. A list, ordered by input burst arrival time, of `qint` objects is maintained to describe all of the bursts currently resident in the queue. The head of the `qint` list describes the earliest burst to arrive, whereas the tail describes the last, or current, input burst to arrive. The queue without flow control can schedule an output `FINISH` message when the appropriate input `FINISH` message arrives. This means that storing details of each resident burst is unnecessary. The flow controlled queue has its output controlled by its consumer, and thus needs details of each input burst it contains. Without storing this, the queue would be unable to properly schedule the output `START` and `FINISH` messages for each burst.

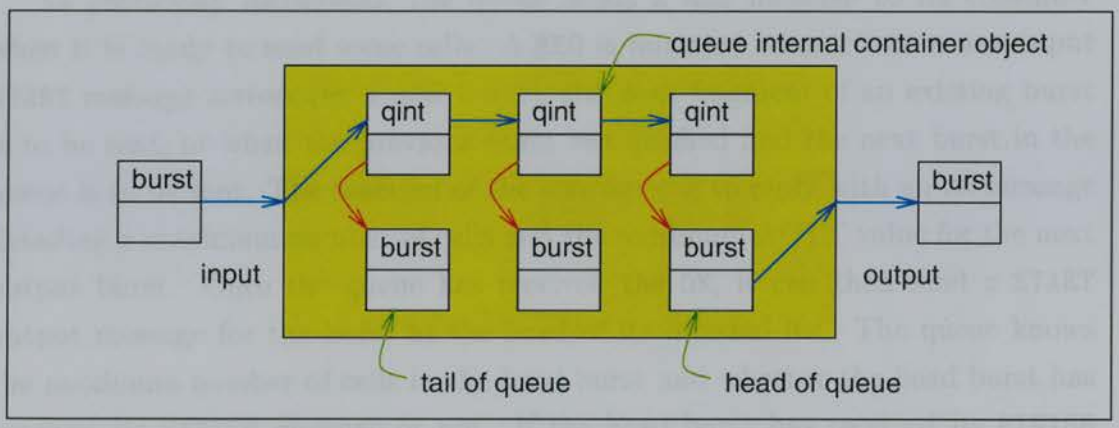


Figure 3.17: The internal structure of the queue with flow control object

3.6.4.1 Queue operation

The four queue scenarios described in Section 3.6.3 are not quite applicable to the flow controlled queue. Instead, the queue keeps track of cell loss by keeping an up to date record of the space remaining in the queue for every event it processes. The idea is that when a burst arrives (ie. the input `START` message), its `qint` object can be passed the free space in the queue at that time. If the queue

sends any bursts during the arrival time of the input burst, the `qint` object has a variable incremented to represent the total number of cells to exit the queue in this time period. The arrival time period of the input burst is the time spanned between the arrival of its `START` and `FINISH` messages. When the `START` arrives, a new `qint` object is created to describe the burst and this becomes the tail of the `qint` list. When the `FINISH` message arrives, the total cell loss for the input burst can be calculated as follows. If the initial space in the queue when a burst arrives is C_S, the total number of cells to exit the queue during the burst arrival is E, and the total number of cells to arrive from the input burst is C_T, then the total loss L for the input burst is:

$$L = C_T - C_S - E$$

If the burst contains more than one cell stream, the cell loss is divided among the individual cell streams by using burst splitting. To ensure that E is an accurate count, a variable representing the last time this value was considered is maintained. Calculating E, relative to this variable, ensures that the exit count in each tail `qint` object represents the number of cells which exit the queue in the arrival period of that burst only.

As previously mentioned, the queue sends a `REQ` message to its consumer when it is ready to send some cells. A `REQ` is initiated when either a new input `START` message arrives (ie. a new burst), the next fragment of an existing burst is to be sent, or when the previous burst has finished and the next burst in the queue is to be sent. The reaction of the consumer is to reply with an `OK` message detailing a maximum number of cells and the minimum ACTT value for the next output burst. Once the queue has received the `OK`, it can then send a `START` output message for the burst at the head of its internal list. The queue knows the maximum number of cells in the head burst and whether the head burst has received its `FINISH` message or not. If the head burst has received its `FINISH` (such that any cell loss has been applied), the number of cells remaining in that burst can be compared with the maximum from the `OK` message. If the head burst has a cell count less than or equal to the maximum, the queue schedules a `START` describing the cells remaining for the consumer, and a `QNEXT` message for itself. If the number of cells remaining is greater than the maximum in the `OK` message, the `START` generated describes a burst of that maximum cell count. If the output `START` is scheduled at T_S, describes C_{max} cells and has an ACTT of $ACTT_{agreed}$ (as determined by the `OK` message), the schedule time for the `QNEXT` message, $T_{\texttt{QNE}}$, is:

$$T_{\texttt{QNE}} = T_S + (C_{max} \times ACTT_{agreed}) \tag{3.23}$$

As long as the minimum ACTT in the OK message is greater than or equal to the queue minimum ACTT, the ACTT for the output burst is set to this value. Otherwise, the output burst ACTT is capped at $ACTT_{min}$ (the minimum for the queue). The QNEXT message is used to trigger the generation of the FINISH message for the current output burst, and also to ensure that the current tail burst has its exit cell count suitably incremented (if it is still arriving). When the QNEXT message arrives, the queue builds and schedules a FINISH message to match the last START sent to the consumer. If the head burst in the list has cells remaining to be sent, another REQ message is scheduled for the consumer. If the head burst has zero cells remaining (ie. the last burst sent to the consumer finished the head burst), the qint object for the head is removed and its successor is promoted to head. If the new head is a qint object, rather than a null pointer, (ie. another burst is resident in the queue) then a REQ is sent to the consumer for that burst. The final act, upon receipt of the QNEXT, is to increment the exit cell count in the tail burst (ie. the number of cells to leave the queue in the arrival period of the tail input burst). This is done by incrementing the exit count (E) by the number of cells sent since the last time E was updated for the tail burst. If the queue is not receiving an input burst when the QNEXT arrives, then the exit cell count is not altered in the tail burst.

In order to update both the exit cell count in the tail burst and the current free space left in the queue, a time variable T_{qc} is used. Every time an output FINISH message is generated by the queue, the free space needs to be incremented and the exit cell count in the tail burst also needs to be incremented (assuming the tail burst is still arriving). The exit cell count in the input tail burst also needs to be incremented (assuming the queue is currently sending an output burst) when the FINISH message for the tail input burst arrives. By setting T_{qc} to the current simulation time when the tail exit cell count is updated, or when the current free space is updated, the number of cells sent by the queue since the previous time either operation was performed can be determined. Thus, when a START message arrives, the current free space recorded in the qint object for the burst will be accurate. When an output FINISH message is generated by the queue, the exit cell count in the tail burst (if it has not finished arriving) will also be accurate.

The behaviour of the queue may change when the head burst is the *only* burst in the queue for which no FINISH has yet been received. If an OK message arrives from the consumer for this burst specifying an ACTT lower than that of the burst,

it is possible that some cells may have queued. This means the queued cells can be sent at the faster rate specified in the OK, rather than the slower arrival rate for the unfinished burst. This is similar to scenario iii (Section 3.6.3) in the queue with no flow control. The first step is to calculate how many cells can be sent from the queue in a burst with an ACTT of $ACTT_c$, the ACTT specified in the OK message. If the arrival time of the OK message is T_{OK}, the arrival time of the head burst is T_{STA}, and the ACTT of the head burst is $ACTT_B$, the number of cells (C_b) which can be sent with an ACTT of $ACTT_c$ can be calculated using Equation 3.24 (which is derived in exactly the same way as Equation 3.22).

$$C_b = \left\lfloor \frac{T_{\text{OK}} - T_{\text{STA}}}{ACTT_B - ACTT_c} \right\rfloor \tag{3.24}$$

However, if C_e cells have already been sent from the head burst, Equation 3.24 becomes Equation 3.25.

$$C_b = \left\lfloor \frac{T_{\text{OK}} - T_{\text{STA}} - (C_e \times ACTT_B)}{ACTT_B - ACTT_c} \right\rfloor \tag{3.25}$$

Equation 3.25 may give a value of C_b which is greater than the number of cells remaining in the head burst. In this case, the value of C_b is set to the maximum number of cells remaining. Once C_b is known, it can be compared to the maximum cell count (C_{OK}) given in the OK message. If there are C_r cells remaining in the head burst, the details of how many cells can be sent in bursts with ACTT values of $ACTT_c$ and $ACTT_B$ is given in Table 3.3.

Condition	Cells sent at $ACTT_c$	Cells sent at $ACTT_B$
$C_b < C_{\text{OK}} \wedge C_{\text{OK}} \leq C_r$	C_b	$C_{\text{OK}} - C_b$
$C_b < C_{\text{OK}} \wedge C_{\text{OK}} > C_r$	C_b	$C_r - C_b$
$C_b \geq C_{\text{OK}} \wedge C_{\text{OK}} < C_r$	C_{OK}	none
$C_b \geq C_{\text{OK}} \wedge C_{\text{OK}} \geq C_r$	C_r	none
$C_b = 0 \wedge C_{\text{OK}} \leq C_r$	none	C_{OK}
$C_b = 0 \wedge C_{\text{OK}} > C_r$	none	C_r

Table 3.3: Number of cells sent at each output ACTT value for an unfinished head burst (assuming $C_b \leq C_r$)

If there are cells which can be sent in a burst with an ACTT of $ACTT_c$, an output START message is created from the head burst describing that number of cells. Rather than schedule a QNEXT message, the queue generates a QCHANGE message for itself. The QCHANGE schedule time is the time when the burst describing the cells sent with an ACTT of $ACTT_c$ should have a FINISH generated. If the output START has a scheduled time of T_S and N_b cells are sent in the burst, the schedule time for the QCHANGE message (T_{QCH}) is:

$$T_{\texttt{QCH}} = T_S + (N_b \times ACTT_c)$$

When the QCHANGE message is received by the queue, an appropriate FINISH message is generated to match the last output START message. The exit cell count in the current queue tail burst is also updated. The next step is schedule another START message if there were any cells to be sent with a burst ACTT of $ACTT_B$, as per Table 3.3. The new output START message is scheduled at $T_{\texttt{QCH}}$ and a QNEXT message is scheduled for the queue at time $T_{\texttt{QNE}}$. If the output burst describes N_B cells with an ACTT of $ACTT_B$ then $T_{\texttt{QNE}}$ is calculated as:

$$T_{\texttt{QNE}} = T_{\texttt{QCH}} + (N_B \times ACTT_B)$$

The arrival of the QNEXT is then treated in the same way as described previously.

The net effect of this approach is to ensure that the queue tries to supply up to the maximum number of cells requested in the OK message. As the ACTT specified in the OK is the *minimum* permitted by the consumer, the queue can send as many cells as it can at the faster rate, and send the remainder with the current input ACTT value.

The operation of the queue, as described, only describes the actions on each "meta" burst. If a burst describes more than one cell stream, burst splitting is used to allocate any total cell counts between each of the component streams. This is used to produce the output START and FINISH messages and to apply cell loss to bursts in the queue. Dealing with the total cell count and burst ACTT values for each burst simplifies the design of the approach. The fine detail is not lost, however, as the decisions made for the burst are applied to its component cell streams when required.

The implementation of the queue with consumer flow control requires a little more complexity as the queue has to be able to cope with the arrival of early input FINISH messages (ie. a burst shorter in actuality than the one described in its START message). If an early FINISH arrives, the queue has to ensure that any QNEXT or QCHANGE messages are rescheduled or cancelled as appropriate to mark the change in the length of the burst. This is achieved by keeping pointers to these events, and using the *reschedule* and *cancel* primitives in the global event list class.

3.7 Switching bursts

The burst switch has the job of routing bursts from one port to another depending on some mapping of bursts to individual switch ports. Rather than design a complex simulation object which can handle multiple `START` and `FINISH` message arrivals on different "virtual" ports, the switch can be modelled with a composite object comprising of multiplexer and demultiplexer objects. Such a composite switch object is shown in Figure 3.18. This will be less efficient than designing a bespoke switch object, as events will need to be generated for every "internal" switch data transfer in the composite design. However, a dedicated switch object would be very complicated to design and produce using the burst-level techniques presented in this chapter.

As can be seen from Figure 3.18, one bidirectional input/output port on the switch is modelled by using a multiplexer and a demultiplexer pair. The demultiplexer is the *input* part of the port, as it can route individual cell streams from incoming bursts to any of its output ports. The multiplexer acts as the *output* part of the port, as it can take bursts from the demultiplexers on any other switch port and merge them to produce output bursts. By suitably connecting the output ports from each demultiplexer to the input ports of each multiplexer, a simple cell switch can be produced.

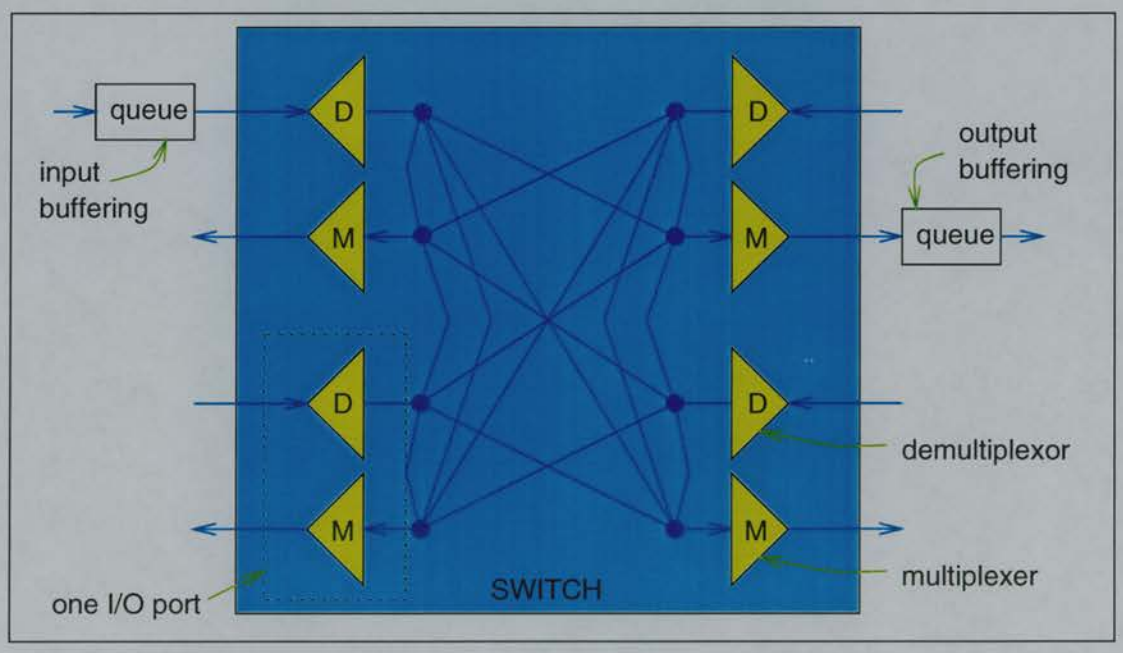


Figure 3.18: A 2x2 compound cell switch object (with optional input and output buffering options shown)

The switch is implemented as a class which takes care of producing the in-

dividual multiplexer and demultiplexer objects, and setting their connections to produce a switch of the dimensions required. The switch object also has a *burst mapping table* which allows for switching of bursts at the individual stream level to any output port. Each cell stream in the simulator can carry a stream identifier and an ultimate destination address. Either value may be used to determine the switch output port for an individual cell stream. Each demultiplexer effectively performs the switching, so the switch burst mapping table is automatically passed to each demultiplexer by the switch object.

Real switching hardware may have either input or output buffering, or both. This can be easily added to the compound switch by connecting suitable queue objects. Input buffering at a switch I/O port requires the queue to be placed before the demultiplexer input port, whereas output buffing places the queue at the output port of the multiplexer. Both options are shown for individual I/O ports in Figure 3.18.

3.8 Summary of chapter

This chapter introduces the integer-time burst-level simulation techniques developed. Firstly, the options for a simulation framework were explored, along with a simple justification of the choice of efficient global event list management techniques used. The indexed event list and the post-ordered tree were found to offer sizeable performance benefits over the simple doubly linked event list, which has the most straightforward operation. However, as the relative performance of each more efficient technique is dependent on the time distribution of the new events added, both have been made available for simulation runs.

The reasons for choosing an all-integer approach were presented. The main advantages are the assured accuracy of integer arithmetic (cf. the floating point alternatives) and the intrinsic speed advantage integer arithmetic has over floating point at the machine level. The choice of simulating at the burst level was also explained. The reason for simulating at an abstracted level of detail is to try to reduce the computational workload required when simulating at the highest level of detail. The burst level is capable of reducing the event count of a simulation when compared with the same model in a cell-level simulator. However, the granularity of the bursts must be carefully chosen to ensure that important behaviour of the system being modelled is not "averaged" out of the results. Care must also be taken that the work necessary to process each event at the burst level does not outweigh the advantage attainable by having less events to process

in the first place.

The notion of a burst in the context of the simulator produced was then described, along with some basic rules for bursts which help simplify the operation of objects within a simulation. Without a strict definition of what constitutes an acceptable burst, simulation objects could produce somewhat inaccurate results due to being too optimistic, or even pessimistic, with the data received. The notion that the `START` message may only contain a *maximum* cell count, for the burst described, helps reduce the margin for error in critical objects such as the multiplexer and queue, for example.

The basic set of simulation objects required to produce simple models were each described in detail. The multiplexer, demultiplexer, queue and switch objects, and their design and function, are crucial to the burst-level simulator. Multiplexing bursts, in particular, will have a large influence on the accuracy achievable by the simulator as it has to produce a single burst capable of describing the possibly many individual cell streams it has to merge. However, the advantage of this technique is that the other simulation objects can use the "meta" burst description to simplify their operation. The fine detail of their behaviour can then be applied over the individual cell streams described within, through the important technique of burst splitting. Burst splitting is useful, as not only can it be used to produce fragments of a burst at some arbitrary split time, but it can also be used to apply cell loss. Cell loss is effectively a split where the split burst fragment is discarded.

Chapter 4

Analysing the core techniques of integer-time burst-level simulation

4.1 Introduction

The purpose of this chapter is look in detail at the accuracy and performance factors of two key techniques in the integer-time burst-level simulator introduced in Chapter 3. Each of the techniques is presented in detail, along with experiments to assess the accuracy implications of their use and the factors which will influence their runtime performance.

The next part of the chapter describes the cost, in terms of compute time, of each type of integer and floating point instruction in a modern microprocessor. In view of the performance penalties incurred from using arithmetic instructions with poor throughput and latency, alterations are suggested for each of the core techniques to reduce their computational costs. This is achieved by replacing integer division instructions with integer division by multiplication of reciprocals. Accuracy and performance results are then presented for each of the core techniques, highlighting the impact of using the revised integer methods.

The final part of the chapter presents a summary of the results, and suggestions for using the revised integer methods.

As described in Chapter 3, the integer time based burst-level simulation technique works by multiplexing streams describing bursts of cells into one multiplexed burst characterised by a single averaged cell transmission time (ACTT) value. Each component stream, which is also a burst containing one or more component streams, is described by a `START` time, a maximum cell count and an ACTT for the burst. The process of producing such a multiplexed burst is the first core technique to be analysed in this chapter, and will be called *burst*

creation. Burst creation is a key technique as it is used in every multiplexer object (Section 3.4), demultiplexer object (Section 3.5) and compound switch object (Section 3.7) used in a simulation. Compound switch objects are comprised of pairs of multiplexer and demultiplexer objects to represent each I/O port, and thus are heavily dependent on the accuracy and performance of the burst creation technique chosen.

Bursts will rarely exist in a simulation lifetime without some alteration due to queuing, loss, link contention or the addition and removal of other cell streams. Section 3.4.3 on page 51 describes the technique of burst splitting which is used to fragment multiplexed bursts in order to simulate cell loss, cell delay variation and the multiplexed addition of other cell streams. The technique of *burst splitting* is the second core technique to be analysed in this chapter. Burst splitting is used in each queue object, multiplexer object (Section 3.6) and compound switch object that models input or output buffers. Burst splitting is a *part* of the burst creation technique as it is used to allocate cells for each component cell stream in a multiplexed burst. Burst splitting is presented separately in the analysis presented in this chapter due to its importance for queue objects. The choice of a burst splitting technique optimal for accuracy and performance will be essential to a simulation using the burst-level techniques.

In the following sections, each of the core techniques is presented along with data showing the accuracy achievable and the factors which limit the performance of the techniques.

4.2 Burst creation

4.2.1 Introduction

Burst creation, as described for the multiplexer in Section 3.4, can be divided into two stages as outlined below.

1. Produce a `START` message with an ACTT value describing the multiplexed burst as well as maximum cell counts for each component stream.

2. Produce a `FINISH` message some period of simulation time *after* the `START` message has been produced. The `FINISH` message contains accurate cell count information for each component stream carried by the multiplexed burst.

The main task of the `START` message production stage is to produce the ACTT value for the burst and this follows the integer technique described in Section 3.4.2

on page 49. The production of the FINISH message uses burst splitting to assign cells from each component stream to the multiplexed burst. This techniques is described in Section 3.4.3 on page 51. Roughly speaking, the two stages required for FINISH message production are as follows.

1. Perform a rough allocation of the cell counts provided by each component stream for the burst.

2. If the time duration of the burst (calculated using the cell stream counts from the first step) is less than the time period required for the burst, extra cells are added the the burst from the component streams which have contributed the largest fraction of a cell. This continues until the time duration is as close as possible to the time period required for the multiplexed burst.

In order to examine the accuracy and performance issues for burst creation, a simple testbed program was developed outwith the main simulation environment. The testbed could be easily manipulated and instrumented to produce data regarding the burst creation technique without the additional overhead of simulation object operation and global event list handling. The testbed program is described in the next section.

4.2.2 Examining the technique in detail

A simple testbed program was produced which contained the code necessary for burst creation. Facilities to provide random number generation, batched experimental runs and result collection were also implemented. The parameters for each burst produced could be set for an experimental run (from a configuration file) so that a range of parameters could be explored. A parameterised number of bursts could be produced so that averaged results could be calculated. The steps taken to produce one burst were as follows.

1. Choose either a random or fixed number of component cell streams to be multiplexed.

2. Assign each component stream an ACTT value chosen from a parameterised random distribution.

3. Assign each component stream a cell count from a parameterised random distribution.

4. Choose the stream with the shortest finish time (ie. *number of cells*$\times$*ACTT of stream*) and use this as the basis for the time period of the multiplexed burst.

5. Calculate the ACTT value for the multiplexed burst and adjust the burst duration so that it is exactly equal to the product of the total number of cells and the burst ACTT.

6. Perform an initial allocation of cells from each component stream in the multiplexed burst using the computed time period and the ACTT value of each component stream.

7. If necessary, perform an allocation of extra cells from component streams so that the time duration of the multiplexed burst is as close as possible to the time period to fill. This stage requires computing cell fractions and then incrementing cell stream counts in descending order of their cell fraction size.

The main areas of investigation for their effects of accuracy and performance of the technique were as follows.

- The use of an *ACTT multiplier* on each component cell stream ACTT value.
- The number of cells in each component burst.
- The number of component bursts multiplexed to produce one burst.

An *ACTT multiplier* is an integer constant by which each component stream ACTT value is multiplied. This is done to increase the size of each ACTT value, to help improve the accuracy of the burst ACTT calculated from the component stream ACTT values. The accuracy is improved due to the range of possible integer values being increased for the burst ACTT (as it is discretely quantised). The larger the ACTT values used to calculate the burst ACTT, the greater the range of values possible for the burst ACTT, thus the value of the burst ACTT will be closer to the ideal value for the multiplexed burst.

To investigate the influence of each of the factors described above, the testbed program recorded the details of each multiplexed burst produced in a program run. The data collected were the total number of cells in the burst, the number of cells in each component stream, the ACTT of the burst and the original ACTT for each component stream.

As a safety check, the integer ACTT value for the burst was also compared against that computed from a floating point based technique calculated as follows.

The ACTT is defined as the time to transmit one cell plus one inter-cell arrival time, so it follows that the reciprocal of the ACTT is the *cell arrival rate* for that stream. Hence, the ACTT for a burst $ACTT_B$ with N component streams, each with an ACTT of $ACTT_i$, can be calculated using Equation 4.1. This value represents the *true* value of the ACTT for the burst rounded up to the nearest integer. This is necessary, as the integer technique rounds the burst ACTT up to the nearest integer.

$$ACTT_B = \left\lceil \frac{1.0}{\sum_{i=1}^{N} \frac{1.0}{(\texttt{double})ACTT_i}} \right\rceil \tag{4.1}$$

4.2.3 Accuracy analysis

To examine the accuracy of burst creation, the final effective ACTT value for each component stream in each burst was calculated. Comparing the effective ACTT for a stream (which has been multiplexed) with its original ACTT gives a measure of the inaccuracy introduced by the technique. Each experiment ignored the effects of multiplexed peak cell rates exceeding the capacity of communication links and the associated problems associated with that situation. For a multiplexed burst M, containing N component streams (each with an original ACTT of $ACTT_i$ and contributing c_i cells to the burst), and with a final ACTT of $ACTT_B$, the time duration of the burst (T_M) is given by Equation 4.2.

$$T_M = \left(\sum_{i=1}^{N} c_i \right) \times ACTT_B \tag{4.2}$$

An effective final ACTT for each component stream ($ACTTf_i$) can then be calculated using Equation 4.3.

$$ACTTf_i = \frac{T_M}{c_i} \tag{4.3}$$

In order to assess the effect of burst creation on the final effective ACTT value for each stream, an *ACTT ratio* is calculated. The ACTT ratio for a stream is defined as the final effective ACTT for the stream (ie. after the stream has been multiplexed) divided by the original ACTT for the stream. Equation 4.4 can be used to calculate the ACTT ratio ($ACTTr_i$) for a stream i. The ACTT ratio is a normalised measure of the inaccuracy introduced to the ACTT of a cell stream. To give a measure of the overall inaccuracy introduced by the burst creation process, the ACTT ratios computed for each cell stream in every burst produced in a testbed program run can be suitably averaged.

$$ACTTr_i = \frac{ACTTf_i}{ACTT_i} \tag{4.4}$$

4.2.4 Performance factors

To generate data for performance analysis of the technique, the testbed code was annotated to count the relative frequencies of integer instructions executed. With some arithmetic instructions being computationally costlier than others (as was shown in Table 3.1 on page 32) it is advantageous to know the mix of instructions executed in the code in order to help optimize it. As well as counting the number and type of arithmetic operations executed, a count of the number of comparison operations was also made. Comparison operations are important, as the result is normally a program branch dependent on the result of the test performed. If many branches are encountered during program execution, the overall performance can be reduced. This is because branches can disrupt the instruction pipelining and dynamic reordering possible in the code, leading to potential execution stalls.

The annotations made to the testbed consisted of explicitly incrementing counter variables for each integer instruction type executed by the dynamically executing program. The number of times each integer instruction type was performed during a run of the testbed program was reported at the end of each run. An average number of integer instructions for each burst produced during a testbed program run could then be presented by dividing the total frequency of each instruction type by the number of bursts produced in each testbed run.

As well as producing instruction execution frequencies for each complete run of the testbed, instruction counts were also made available for certain parts of the burst creation technique. Specifically, counts were made of the integer instruction types executed for ACTT calculation and the addition of cells based on fractions. As the addition of cells based on fractions is dependent on whether the initial allocation of cells to the multiplexed burst fills the required time period or not, the number of instances where this was necessary was also recorded.

Like the simulation environment described in Chapter 3, the testbed program used an unsigned 64-bit integer variable (type `unsigned long long int` in C++) to hold the stream ACTT values. This variable was set with a C++ `typedef` statement to be type `time_type`. The types of instruction counted in the annotated testbed program were as follows:

Mult Integer multiplication
(eg. `time_type c = (time_type)a * (time_type)b;`)

Div Integer division

(eg. `time_type c = (time_type)a / (time_type)b;`)

Plus Integer addition

(eg. `time_type c = (time_type)a + (time_type)b;`)

Minus Integer subtraction

(eg. `time_type c = (time_type)a - (time_type)b;`)

Compare Integer comparison

(eg. `if((time_type)a >= (time_type)b))`

Measurements of testbed program run time were not recorded for these experiments due to the nature of the simple testbed program produced. As the complete simulator requires the support of a discrete event list handler, it makes more sense to produce data on the performance runtimes when the techniques described here are incorporated into the total environment (see Chapter 5). However, making the core techniques as fast and efficient as possible will benefit the runtime performance of the complete simulator. The simple testbed used in the experiments performed in this chapter allowed for such optimisations to be explored.

4.2.5 First accuracy results

As the first investigation, the simple testbed was given some random values for the number of streams to multiplex, the number of cells per stream and the ACTT value of each stream. Four different ACTT multipliers were used for each experiment. Table 4.1 gives the values used in the first two batches of experiments. The distribution *Randint* is defined as $x = Randint(a, b)$ where x is a random integer in the range $a \leq x \leq b$ and where each possible x has equal probability of selection.

Parameter	Batch 1 ranges	Batch 2 ranges
Number of runs	100000	100000
ACTT multipliers	1, 10, 100 & 1000	1, 10, 100 & 1000
Number of streams	$Randint(2, 20)$	$Randint(2, 20)$
ACTTs of streams	$Randint(10, 1000)$	$Randint(10, 1000)$
Number of cells	$Randint(10, 1000)$	$Randint(100, 10000)$

Table 4.1: Values for first experimental runs

The first check was of the accuracy of the burst ACTT value calculated by the integer technique compared to that calculated by the floating point technique

(see Section 4.2.2). There were no disagreements between the integer and floating point results for any of the experiments in both batches.

Figure 4.1 shows the average ACTT ratio for each ACTT multiplier used in the experiments along with the minimum and maximum ACTT ratios encountered in each experimental run. The average ACTT value was used to give an overview of the degree of inaccuracy introduced to the ACTT values for the experimental parameters chosen, and was calculated by summing the ACTT ratio values for every component stream in the entire testbed program run and dividing by the total number of component streams (ie. the sum of the number of component streams in each multiplexed burst produced in the program run). Table 4.2 gives the numerical results for the results shown in Figure 4.1. Figure 4.2 on page 87 shows the spread of ACTT ratios for each of the ACTT multiplier values used in the experiments.

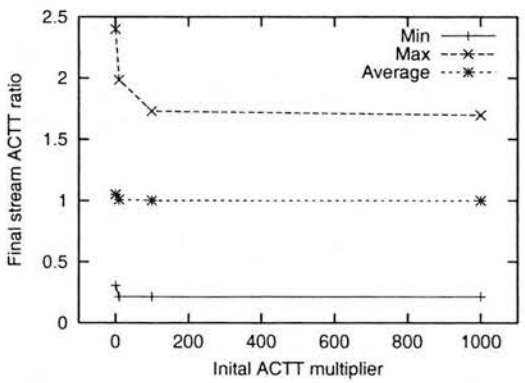
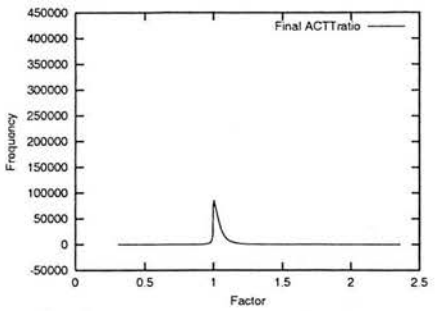


Figure 4.1: Minimum, average and maximum ACTT ratio results from experiment Batch 1

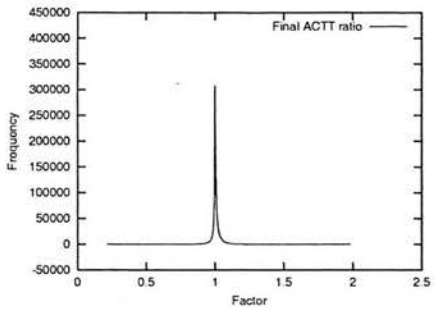
ACTT Multiplier	Min	Max	Av.	sd	Average burst size (in cells)
1	0.303797	2.40000	1.05004	0.0823498	852
10	0.213811	1.98545	1.00870	0.0495778	854
100	0.213811	1.72976	1.00080	0.0451877	856
1000	0.213811	1.69769	0.99994	0.0449891	856

Table 4.2: Data for Figure 4.1

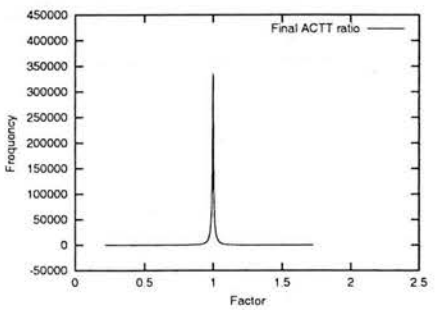
Figure 4.3 shows the minimum, average and maximum final stream ACTT ratios for the experiments in Batch 2 (the numerical results are shown in Table 4.3). The spread of ACTT ratio values for each of the ACTT multipliers used in the experiments is shown in Figure 4.4. Batch 2 differed from Batch 1 in that each component burst stream had a greater number of cells (as the cell value was chosen from a random distribution with a greater range of values).



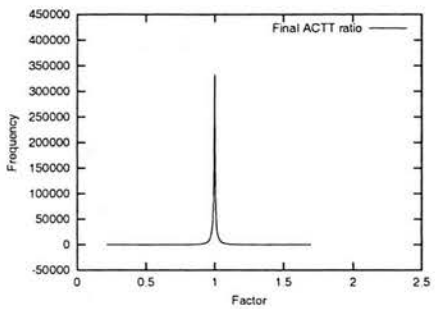
(a) ACTT multiplier = 1



(b) ACTT multiplier = 10



(c) ACTT multiplier = 100



(d) ACTT multiplier = 1000

Figure 4.2: Spread of final stream ACTT ratios for experiment Batch 1

4.2.5.1 Accuracy Analysis

Considering the integer technique used, one would expect that as the initial ACTT multiplier increases, the average final ACTT ratio would get closer to the ideal value of 1 (ie. the case where the multiplexing technique introduces no error to the ACTT values of the component streams). This is due to the *resolution* of the ACTT value for the multiplexed burst increasing with increasing ACTT magnitude. As the burst ACTT is rounded up to the nearest integer value, the burst ACTT computed from component ACTT values with a small magnitude is likely to be an *overestimate* of the ideal value. For example, if the ideal floating point value for the burst ACTT of some burst was 3.1734 (as could be computed with Equation 4.1 before the integer ceiling is applied), the accuracy of the integer ceiling would vary with the range of values available for the integer representation. Rounding the burst ACTT up to 4 would be a large overestimate of the ideal burst ACTT. If each component ACTT used to calculate the burst ACTT were initially multiplied by 10, the ideal burst ACTT would be 31.734. The integer ceiling representation, 32, is a lesser overestimate of the ideal value. Increasing

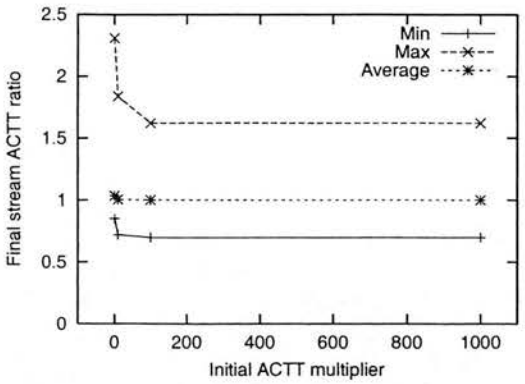


Figure 4.3: Minimum, average and maximum ACTT ratio results for experiment Batch 2

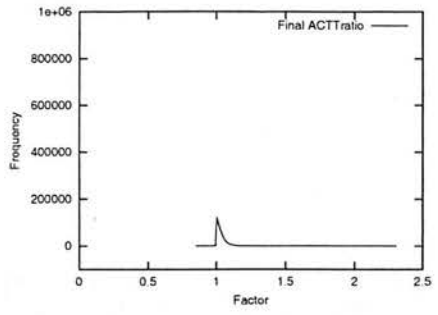
ACTT Multiplier	Min	Max	Av.	sd	Average burst size (in cells)
1	0.849673	2.30882	1.03479	0.03448430	8584
10	0.719810	1.83961	1.00490	0.00952668	8584
100	0.697602	1.62305	1.00080	0.00648364	8587
1000	0.697212	1.62150	1.00002	0.00627719	8589

Table 4.3: Data for Figure 4.3

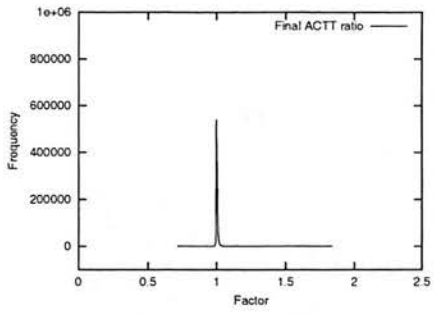
the constant ACTT multiplier used for each component stream increases the resolution of the burst ACTT value computed, and so lowers the overestimate of the ideal value.

The accuracy of the computed integer burst ACTT also has direct bearing on the accuracy of the burst *duration*, which is the timespan necessary to carry the number of cells described by the burst (ie. the product of the the number of cells and the burst ACTT). Any overestimate in the burst ACTT value will result in an overestimate of the time duration of the burst. The initial cell allocations for each component stream are calculated from the duration of the multiplexed burst, so are dependent on the accuracy of the burst ACTT. The larger the overestimate in the burst ACTT, the larger the potential cell count returned for each stream. This may lead to an overestimate of the number of cells to carry in the burst for a stream and so a less accurate final effective ACTT for the stream. The effective ACTT values of the component cell streams will be closer to their original ACTT values when the burst timespan is optimal.

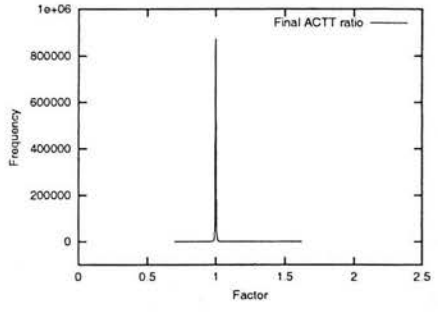
It is also expected that as the number of cells described by a multiplexed burst increases, the accuracy of the ACTT ratios for each component stream will also increase. This is because unless a multiplexed burst describes a time duration equal to the *minimum sequence time* for the cell streams described, component cell streams may describe time periods smaller than the duration of the entire



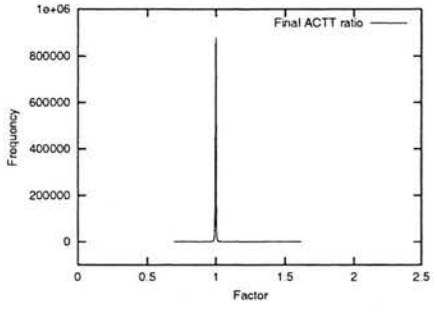
(a) ACTT multiplier = 1



(b) ACTT multiplier = 10



(c) ACTT multiplier = 100



(d) ACTT multiplier = 1000

Figure 4.4: The ACTT ratio results for experiment Batch 2

burst (ie. the multiplexed burst duration will describe an integral number of cells plus some empty "padding" time for each component stream). The size of the time padding for each component depends on the inaccuracy of the burst ACTT and the cell allocation to individual streams within the burst. If the burst ACTT is inaccurate (as is expected for small ACTT multipliers), the duration of the burst will be larger than the required duration, leading to larger padding times. The padding time represents an error introduced by the technique, but the magnitude will be averaged over the number of cells carried in the stream. Hence, the larger the number of cells carried in a burst, the lower the final ACTT ratio should be.

The expected improvements in final ACTT ratio described above are reflected in the results. Figure 4.2 (and Table 4.2) show how the average ACTT ratio, and its standard deviation, improve with increasing ACTT multiplier for the experiments in Batch 1. The expected improvements in the average ACTT ratio for bursts describing a greater number of cells are shown in Figure 4.4 (and Table 4.3) which are the results from the experiments in Batch 2. Once again, increasing the ACTT multiplier makes the average ACTT ratio closer to 1 (while reducing the standard deviation), and decreases the range of ACTT ratios produced in each

experiment.

To assess the effect of different ACTT multiplier values on the duration of each burst, the testbed recorded whether the first allocation of cells to each burst produced an underestimate or overestimate of the timespan to be filled. Table 4.4 shows the distribution of bursts which had a time period greater than, equal to or less than the required burst time duration after the first allocation of cells for the experiments in Batch 1. As expected, the percentage of overestimates decreases with increasing ACTT multiplier due to the accuracy of the burst ACTT improving. The relative difference in minimum to maximum ACTT ratio spreads seen in Figures 4.2(a) and 4.2(d) show the benefit of using a large ACTT multiplier. When there is an overestimate at the first allocation of cells, the burst is longer than it should be, so it is more likely that the final effective ACTT for each component will also be an overestimate. Greater accuracy in the burst ACTT leads to an overall improvement in the accuracy of the final effective ACTT ratios for each of the component streams as shown by the narrowing spread of the distribution of ACTT ratios with increasing multiplier.

ACTT multiplier	% Greater than	% Equal to	% Less than
1	68.874	7.420	23.706
10	8.657	6.742	84.601
100	0.001	0.288	99.711
1000	0.000	0.067	99.933

Table 4.4: Percentage of multiplexed bursts which are $>$, $=$ or $<$ the required time period after the first allocation of cells for Batch 1

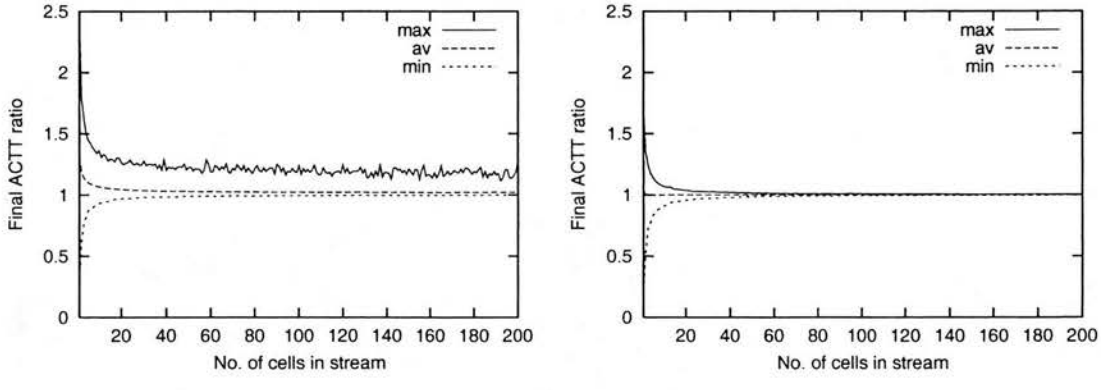
The effect of the burst duration overestimate is also seen in the results from Batch 2, where there were more cells per burst than in Batch 1. As expected, the lower ACTT multipliers lead to a greater percentage of first allocation overestimates as shown in Table 4.5. There are also comparatively more overestimates in Batch 2 than for the same ACTT multipliers in Batch 1. This is due to having more cells to consider at the first allocation of cells where the accuracy is heavily dependent on the accuracy of the burst ACTT. With more overestimates, the final effective ACTT ratios are also likely to be overestimates. This effect is shown in the relative differences between the ACTT ratio lower bounds in Figure 4.1 and Figure 4.3.

Figure 4.1 and Figure 4.3 also show that the range of ACTT ratios can be large, even if the average ACTT ratio is close to 1 with a small standard deviation. This is due to the error in each component stream ACTT being averaged over the number of cells carried in the stream. Hence, for an increasing number of cells in

ACTT multiplier	% Greater than	% Equal to	% Less than
1	95.944	1.368	2.688
10	69.268	7.286	23.446
100	8.913	6.729	84.358
1000	0.001	0.299	99.700

Table 4.5: Percentage of multiplexed bursts which are $>$, $=$ or $<$ the required time period after the first allocation of cells for Batch 2

a stream, the difference in the minimum to maximum ACTT ratio range should decrease (as well as decreasing with increasing ACTT multiplier). Figure 4.5 shows the ACTT ratio range for cell streams containing between 1 and 200 cells in Batch 1 for ACTT multipliers of 1 and 1000. As can be seen from the figure, the range of values is reduced when each component cell stream describes a larger number of cells. The greater number of burst overestimates at the first cell allocation when low ACTT multipliers are used can be seen in the non-converging maximum ACTT ratio plot in Figure 4.5(a).



(a) ACTT mult = 1

(b) ACTT mult = 1000

Figure 4.5: Range of ACTT ratios for cell streams containing between 1 and 200 cells for Batch 1

4.2.5.2 Performance factors

As described in Section 4.2.4, the testbed program used in these experiments was instrumented to count the integer instruction types executed in the process of burst creation. Figure 4.6 (on page 96) shows the average number of integer instructions per multiplexed burst produced in each of the experimental batches. As can be seen from Figures 4.6(a) and 4.6(b), which show the average instruction counts for the entire burst creation process, the averages are slightly lower

for Batch 2. Looking in detail at which part of the process is responsible for this, Figures 4.6(e) and 4.6(f) show that the Batch 2 experiments require less instructions to be executed per burst when considering the addition of cells based on fractions. This is due to the number of bursts which require extra cells as the first cell allocation producing a burst shorter than the required time period. Comparing the "% Less than" columns in Tables 4.4 and 4.5 shows that having a greater number of cells reduces the need for cell addition based on fractions, with this effect reducing with increasing ACTT multiplier. An ACTT multiplier of 1 in both batches produces the highest fraction of streams which overestimate the time period to fill at the first cell allocation. This means that few cell additions based on fractions are required, hence the reason for the the lowest instruction count averages when the multiplier is 1.

The instruction frequency results for burst ACTT calculation (Figures 4.6(c) and 4.6(d)) are very similar, as the ACTT values for the component cell streams only differ between the batches by a fixed constant (ie. the ACTT multiplier) with all of the other factors the same. As can be seen, the method used to calculate the burst ACTT is computationally expensive, owing to the large number of integer divides which incur the highest computational cost. Any means of reducing the number of integer division counts in the ACTT calculation may help improve performance (see Section 4.4).

4.2.6 Accuracy and performance analysis for varying numbers of component streams

After performing the first two batches of experiments to look at the accuracy and performance of the burst creation technique, the next step was to look at the effect of varying the number of component streams. Table 4.6 shows the parameters chosen for experimental batches 3 and 4. Rather than perform the analysis for each of the four original ACTT multipliers, two were chosen to show the range of results available. From the first experiments, it is clear that the accuracies improve with increasing ACTT multiplier, with a multiplier of 10 giving a large accuracy advantage over a multiplier of 1. Hence, 10 was chosen as the first of the ACTT multipliers to be examined. There is little difference in the results obtained from multipliers of 100 and 1000, so 1000 was chosen to provide a contrast to the results when the multiplier is 10.

Parameter	Batch 3 ranges	Batch 4 ranges
Number of runs	100000	100000
ACTT multipliers	10 & 1000	10 & 1000
Number of streams	{2,3,...,20}	{2,3,...,20}
ACTTs of streams	$Randint(10, 1000)$	$Randint(10, 1000)$
Number of cells	$Randint(10, 1000)$	$Randint(100, 10000)$

Table 4.6: Parameters for the 3rd and 4th experimental runs

4.2.6.1 Accuracy analysis

Once again, the first check performed was of the accuracy of the ACTT value computed by the integer technique compared to the value computed by the floating point technique calculated with Equation 4.1 (see Section 4.2.2). As with the previous experiments, there were no differences recorded in any of the experiments.

Figure 4.7 (on page 97) shows the average final ACTT ratio for each ACTT multiplier in each batch. The results confirm what was seen in experimental batches 1 and 2, namely the improvement in the average and standard deviation of the final ACTT ratio with increasing ACTT multiplier. Once again, increasing the number of cells improved the final ACTT ratio results as the error is spread over the number of cells described by the burst. However, the results show that the final ACTT ratio diverges from the ideal value of 1 as the number of cell streams increases. The ACTT ratio standard deviation also increases with the increasing number of streams. This is a side effect of the accuracy of the calculated burst ACTT value and the fact that each burst may not describe the minimum sequence time for the streams contained. The combined effects of these sources of error are magnified when more cell streams are included in the final multiplexed burst, as more streams have to be considered for the time period to be filled. This results in a greater number of inexact final ACTT ratios to be considered at the analysis stage. The greater the accuracy of the burst ACTT (which improves with increasing ACTT multiplier), the smaller the variation of the final ACTT ratios. This is demonstrated in the differences between the results where the multiplier is 1 and when it is 1000 in Figure 4.7.

Figure 4.8 (on page 97) shows the relative percentages of streams greater than, equal to or less than the required burst time period after the first cell allocation when the ACTT multiplier is 10 in both batches. As in batches 1 and 2, the case where the multiplier is 1000 resulted in nearly 100% of all streams being less than the required period in batches 3 and 4. Hence, these results are not shown. As can be seen from Figure 4.8, as the number of cells increases the

percentage of overestimates also increases. The number of overestimates also increases with increasing number of component streams. This is as expected, due to the inaccuracy of the burst ACTT when the ACTT multiplier is 10, coupled with the increasing number of streams making it more likely that an overestimate will occur. When there are more cells, the error is magnified and the chances of an overestimate increase as shown in Figure 4.8(b). This matches the behaviour shown in batches 1 and 2.

4.2.6.2 Performance factors

Figure 4.9 shows the average integer operation counts for the experiments in batches 3 and 4. The average is taken as the average number of operations per multiplexed burst produced. Once again the variation in the averages is mainly due to the effect of adding cells based on fractions. In the cases where the multiplier is 10 (Figures 4.9(a) and 4.9(c)), the difference in instruction averages over the whole process is clear. This is due to the extra demands of adding cells by fractions when the ACTT multiplier is 10. Figure 4.10 shows the difference in the average number of instructions per burst, for adding cells by fractions, required for an ACTT multiplier of 10 for batches 3 and 4. The Batch 3 experiments (Figure 4.10(a)) require more instructions due to adding cells by fractions, hence the relative increase in average instruction counts in Figure 4.9(a) over Figure 4.9(c).

The difference in adding cells by fractions between batches 3 and 4 is due to the relative frequency of burst length duration overestimates after the first cell allocation. This is related to both the number of cells that the burst describes and the ACTT multiplier used. For the experiments where the ACTT multiplier was 1000, nearly 100% of first allocations were underestimates, thus making the instruction averages virtually identical.

The integer instruction averages for the burst ACTT value calculation remained unchanged between the experiments, as would be expected. As was the case in batches 1 and 2, the integer ACTT calculation was chiefly responsible for the high number of integer divide operations in the overall instruction averages. This made burst ACTT calculation the most computationally expensive part of the burst creation process with the cost increasing with the number of burst streams merged to make each burst, as would be expected.

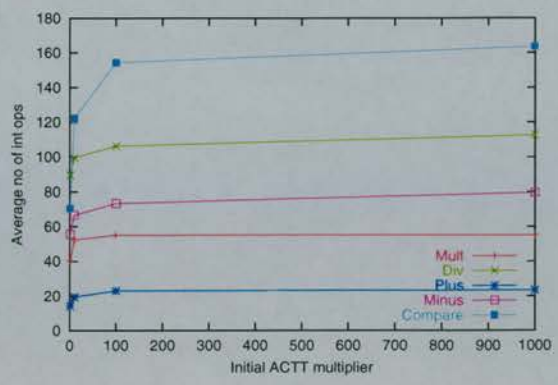
4.2.7 Summary

The experiments presented in the previous sections were designed to give an overview of the accuracy and performance implications of the burst creation pro-

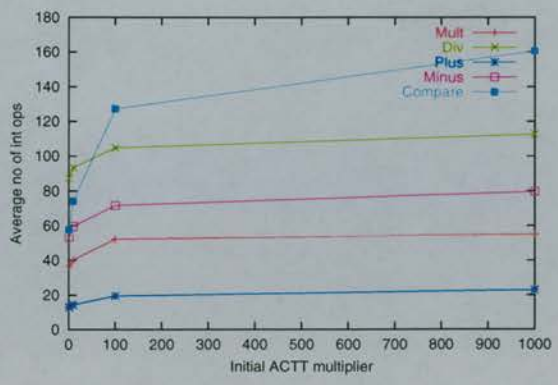
cess. The notion of an ACTT multiplier was introduced, and the use of such a technique was shown to reduce the average final ACTT ratios for cell streams when they are multiplexed to produce a single burst. The higher the ACTT multiplier chosen, the better the average ACTT ratio obtained. The standard deviation of the average ACTT ratio, as well as the spread of final ACTT ratios observed, also decreased with increasing ACTT multiplier. The effect of increasing the number of cell streams multiplexed was shown to decrease the accuracy of the average ACTT ratio. However, the choice of a large ACTT multiplier reduced the inaccuracy in the final average ACTT ratio for an increasing number of cell streams.

In terms of the number of integer operations required, the calculation of the burst ACTT was shown to be the major contributor of computationally expensive integer divide operations to the whole process. Increasing the number of cell streams multiplexed increased the average number of integer operations required per burst, as would be expected.

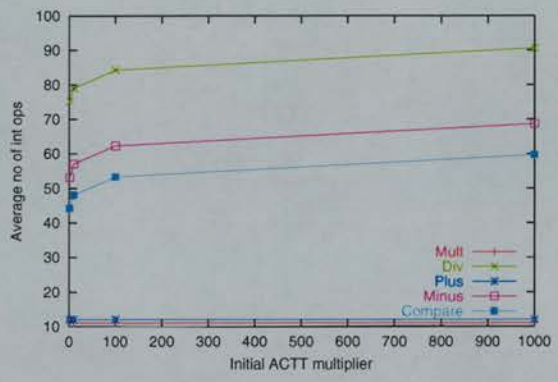
Differences in the average number of integer operations required for different ACTT multipliers were due to the requirement to add cells by fractions. This is necessary when the first allocation of cells from the component streams to a multiplexed burst produces a burst which underestimates the time period to be filled. The percentage of bursts requiring cells added by fractions is related to the accuracy of the burst ACTT (which is related to the ACTT multiplier chosen) and the number of cells in the burst. A small ACTT multiplier results in a burst ACTT which is an overestimate. If the burst describes a large number of cells, the first allocation of cells will mostly produce bursts which are overestimates of the time period to fill. For smaller bursts, the first allocation of cells mainly produces bursts which are underestimates of the time period, and thus require cells to be added by fractions. For larger ACTT multipliers, the burst ACTT is more accurate, so most bursts produced after the first allocation of cells are underestimates of the time period to fill. This requires cells to be added by fractions for nearly every burst produced.



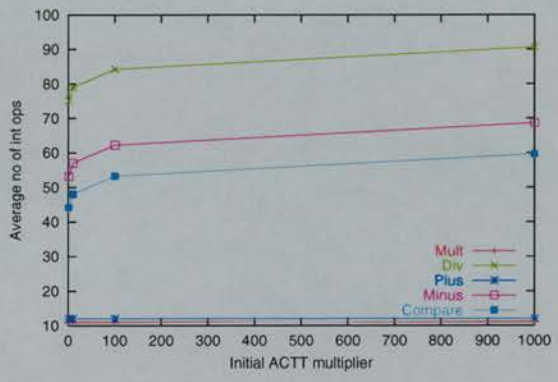
(a) All integer operations for Batch 1



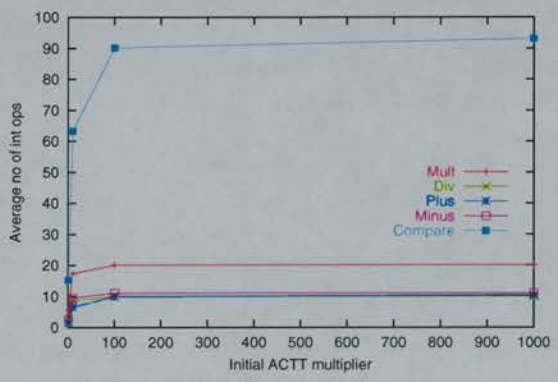
(b) All integer operations for Batch 2



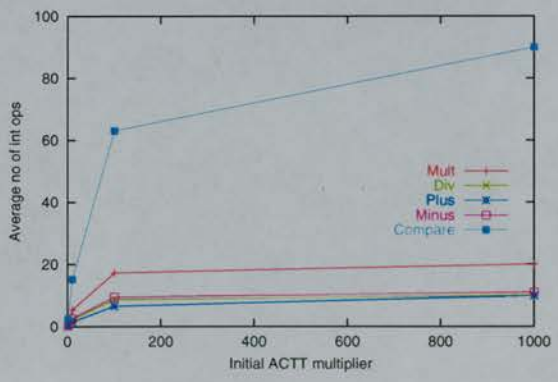
(c) ACTT int ops for Batch 1



(d) ACTT int ops for Batch 2

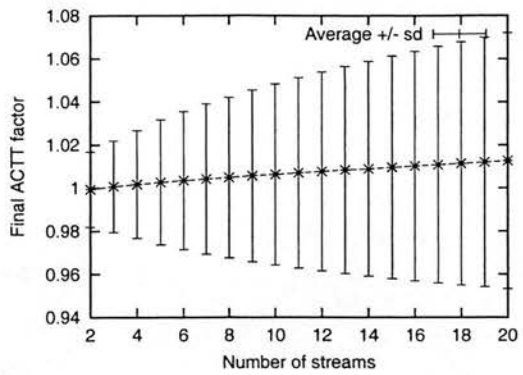


(e) Cell addition int ops for Batch 1

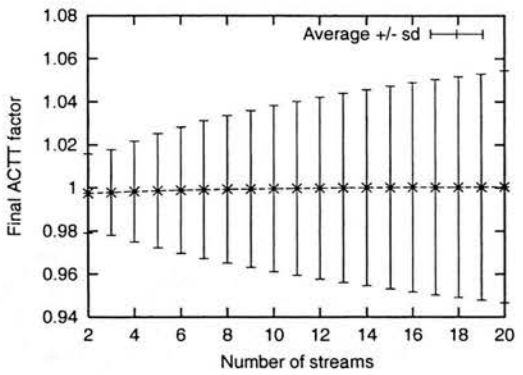


(f) Cell addition int ops for Batch 2

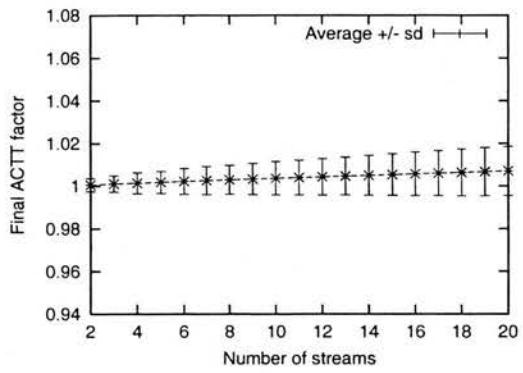
Figure 4.6: Integer operations averaged over the number of multiplexed bursts produced



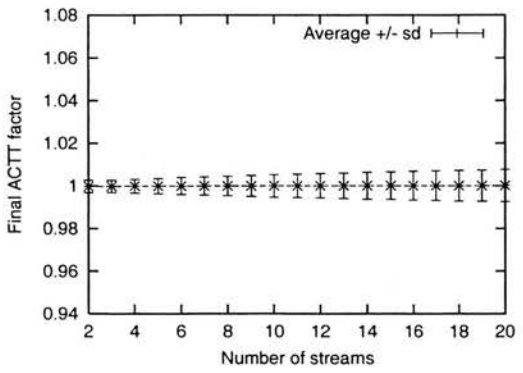
(a) Batch 3 (ACTT=10 cells=10,1000)



(b) Batch 3 (ACTT=1000 cells=10,1000)

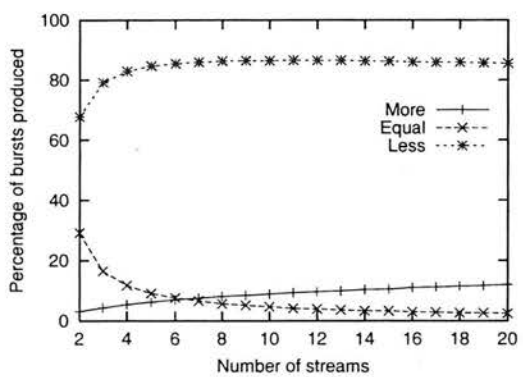


(c) Batch 4 (ACTT=10 cells=100,10000)

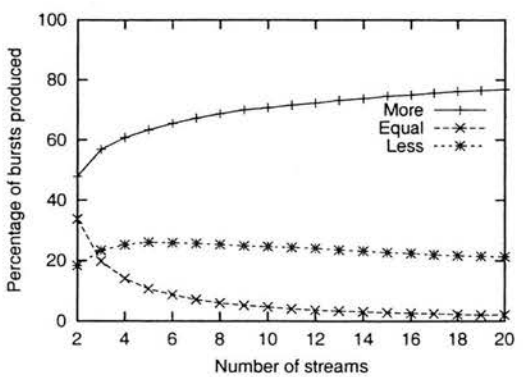


(d) Batch 4 (ACTT=1000 cells=100,10000)

Figure 4.7: Average final ACTT ratios (with standard deviations) for experiment batches 3 and 4

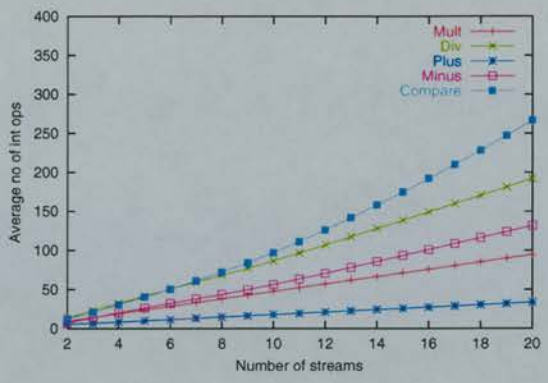


(a) Batch 3 (streams of 10-1000 cells)

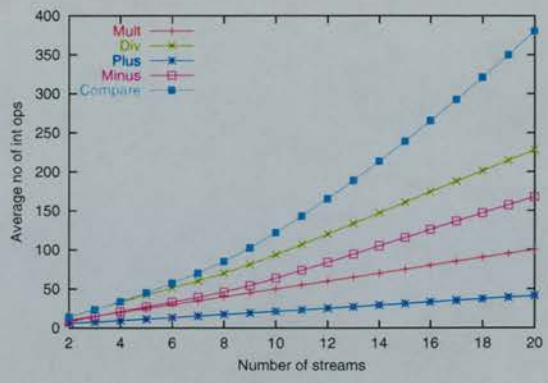


(b) Batch 4 (streams of 100-10000 cells)

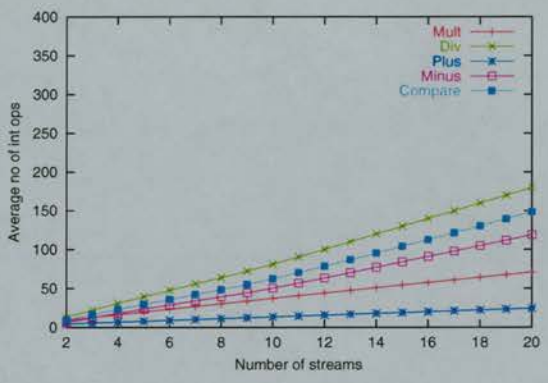
Figure 4.8: Relative percentages of bursts produced which were >, = or < the time period to fill when the ACTT multiplier was 10



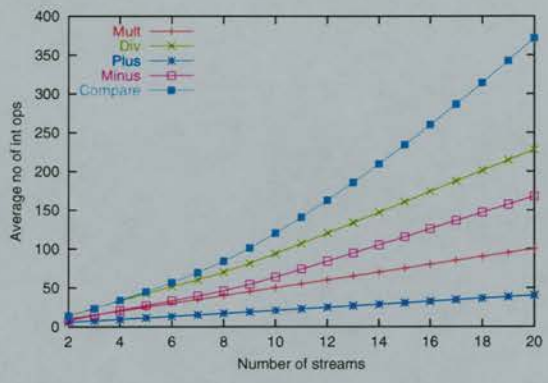
(a) Batch 3 (ACTT=10 cells=10,1000)



(b) Batch 3 (ACTT=1000 cells=10,1000)

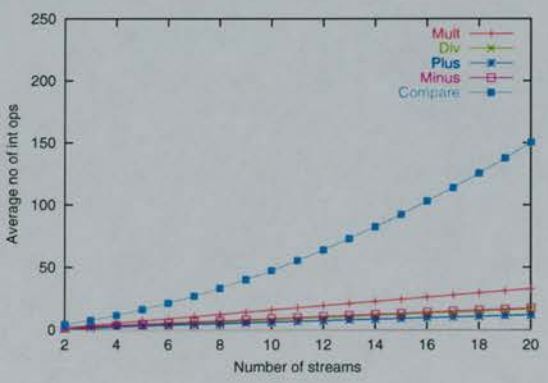


(c) Batch 4(ACTT=10 cells=100,10000)

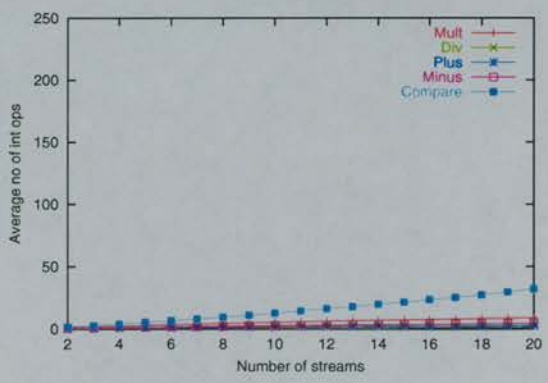


(d) Batch 4 (ACTT=1000 cells=100,10000)

Figure 4.9: Average number of integer instructions per burst produced in experiment batches 3 and 4



(a) Batch 3 (ACTT=10 cells=10,1000)



(b) Batch 4 (ACTT=10 cells=100,10000)

Figure 4.10: Average number of integer instructions per burst for cell addition based on fractions

4.3 Burst splitting

4.3.1 Introduction

The second core technique to be examined is that of burst splitting. Burst splitting essentially splits one burst into two time sequential bursts, ensuring that the cell counts in each split burst "fragment" correlate to the ACTT values of each component stream. The technique is used to ensure that the number of cells described by a multiplexed burst reflects the individual characteristics of the component cell streams being merged. Burst splitting is also used to allocate cell loss over the component streams in a multiplexed burst which encounters cell loss. As burst splitting is an essential part of the multiplexing and queueing burst-level operations, it is widely used in the simulator.

In the case of the multiplexer (see Section 3.4), the bursts produced at its output port are based on the time coincident bursts at its inputs. When a burst is created, a `START` message is produced and an appropriate `FINISH` message is produced when the state of one of the inputs changes (ie. a new burst `START` message arrives or a `FINISH` message arrives for an input burst). When this happens, the multiplexer calculates a time period for the current multiplexed burst and then uses burst splitting to calculate the cell count contribution of each input cell stream to produce the `FINISH` message for the burst. The splitting function is given the time period to be filled, the ACTT of each input cell stream and the maximum number of cells each cell stream can contribute. It is essential that cell counts for each component stream do not exceed their maximum values.

The queue objects (see Section 3.6) also use burst splitting to calculate the component stream cell contributions every time there is a change of state in the queue output, and when cell loss needs to be calculated. Cell delay variation is modelled as a change in the burst ACTT value at the queue output, so burst splitting is used to determine the cell counts of the component streams in each burst fragment. Cell loss is modelled as the removal of cells from the burst and burst splitting is used to determine the allocation of the lost cells amongst the component streams.

Broadly speaking, the stages of burst splitting (as described in Section 3.4.3), based on filling a time period with cells from an existing burst, are as follows:

1. Calculate the number of cells for the required time period using the burst ACTT.

2. Perform an initial allocation of cells from each component stream in the burst based on their ACTT values.

3. Check if the total number of cells computed at the first allocation matches the number calculated in step 1.

   - If there is a shortfall, use cell contribution fractions to increment component stream cell counts until the target number is achieved.
   - If there is an over-estimate, scale the contribution counts for each component stream. If this does not produce the target cell count, increment counts using cell contribution fractions.

Cell counts from each stream have to be *scaled down* if the first allocation of cells in the split procedure produces too many cells. For a split time period T_S, for a burst with an ACTT of $ACTT_B$ and N component cell streams, the maximum number of cells to assign to the split time period, C_S, is:

$$C_S = \left\lceil \frac{T_S}{ACTT_B} \right\rceil$$

The first cell count allocation, c_i, from component stream i with an ACTT value of $ACTT_i$ is:

$$c_i = \left\lfloor \frac{T_S}{ACTT_i} \right\rfloor$$

Hence the total number of cells allocated to the split time period after the first allocation of cells, C_{first}, is:

$$C_{first} = \sum_{i=1}^{N} c_i$$

If $C_{first} > C_S$, then the component cell counts need to be scaled down as too many cells have been allocated. The scaled cell count for a stream, c_i' is calculated with:

$$c_i' = \left\lfloor \frac{C_S}{C_{first}} \times c_i \right\rfloor$$

The scaled cell count for stream i can then be used subsequently for the cell allocation of that stream to the split time period. If the addition of cells based on fractions is required to ensure that a total cell count of C_S is reached, the scaled cell counts can be used in the fraction calculations.

4.3.2 Examining the technique

In order to examine the technique of burst splitting, the simple testbed program used to examine burst creation (Section 4.2.2) was modified such that the bursts

produced could be split at an arbitrary point in time in their duration. The choice of extending the previous testbed was made, so that experiments could be performed with the same parameters as used in the burst creation experiments. This allowed comparison of the effects of different ACTT multipliers and of varying numbers of component streams in each burst. However, using this experiment to determine the accuracy impact of burst splitting is not meaningful. This is because the burst split produces two bursts sequential in time (ie. the `START` message for the second burst would be at the same simulation time as the `FINISH` message from the first burst), each with the same burst ACTT value. Averaging the final effective ACTT for each component cell stream over the duration of the two bursts will yield the same value as the final effective ACTT ratio over the original burst. Burst splitting will produce final ACTT ratio differences for each cell stream in each fragment, but averaging the fragments removes this error. It is only when split bursts are multiplexed with other bursts that the accuracy impact of the approach can be noted, and this is explored in more detail in the experiments in Chapter 5.

The testbed was annotated to provide the integer instruction execution counts for the technique to assess the performance factors. As this technique has to ensure a fair cell count for each component stream in a split burst, it is important that the accuracy impact of the algorithm used, and the factors influencing the performance, can be ascertained. To this extent, the testbed also recorded how capable the technique was at filling the time period for each split it was asked to perform (ie. the total number of cells allocated at the first allocation, compared to the maximum number for the split time period).

4.3.3 Experimental results and analysis

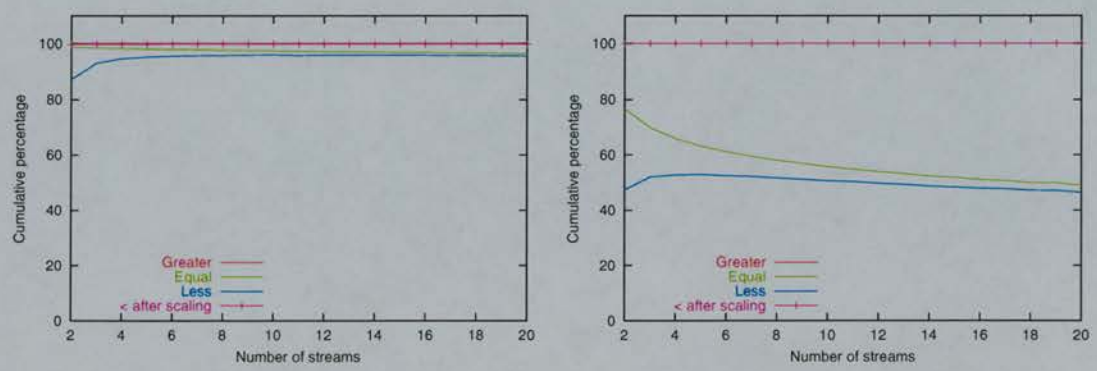
Table 4.7 shows the values used in the two batches for the burst splitting experiments. The distribution *Uniform* is defined as $x = Uniform(a, b)$ where x is a random double precision floating point number in the range $a \leq x \leq b$, where each x has equal probability of selection. For the same reasons as in the burst creation experiments, the ACTT multipliers chosen were for the burst splitting experiments were 10 and 1000. No experiments were performed for ACTT multiplier values of 1 and 100. The number of cell streams used to build each burst to be split was varied between 2 and 20.

Figure 4.11 shows a cumulative percentage graph representing how well, after the first allocation of cells, the burst splitting technique matched the required number of cells for the split when the ACTT multiplier was 10. The percentage

Parameter	Batch 5	Batch 6
Number of runs	100000	100000
ACTT multiplier	10 & 1000	10 & 1000
Number of streams	{2,3,...,20}	{2,3,...,20}
ACTTs of streams	*Randint*(10, 1000)	*Randint*(10, 1000)
Number of cells	*Randint*(10, 1000)	*Randint*(100, 10000)
Split (as % of burst duration)	*Uniform*(0.0, 100.0)	*Uniform*(0.0, 100.0)

Table 4.7: Values for the burst splitting experimental runs

of bursts under the "Equal" line (including those "Less" than the required number of cells) match the required cell count *exactly* for the split at the end of the burst split procedure. The addition of cells based on cell fractions is required when the burst cell count is "Less" than the required number of cells for the burst after the first allocation. The percentage of streams between the "Greater" and "Equal" lines represent the number of cell count overestimates which have to be *scaled down* to match the required number of cells. The "< after scaling" line shows the number of overestimate bursts which have cell counts less than the required number of cells for the split after being scaled down. The results are not shown for the case when the ACTT multiplier is 1000 as the percentage of split bursts less than the time period to be filled is nearly 100% in every case.



(a) Batch 5 (ACTT=10 cells=10,1000)

(b) Batch 6 (ACTT=10 cells=100,10000)

Figure 4.11: Cumulative percentages of how split bursts compared to the size of split required after the first allocation of cells to the split

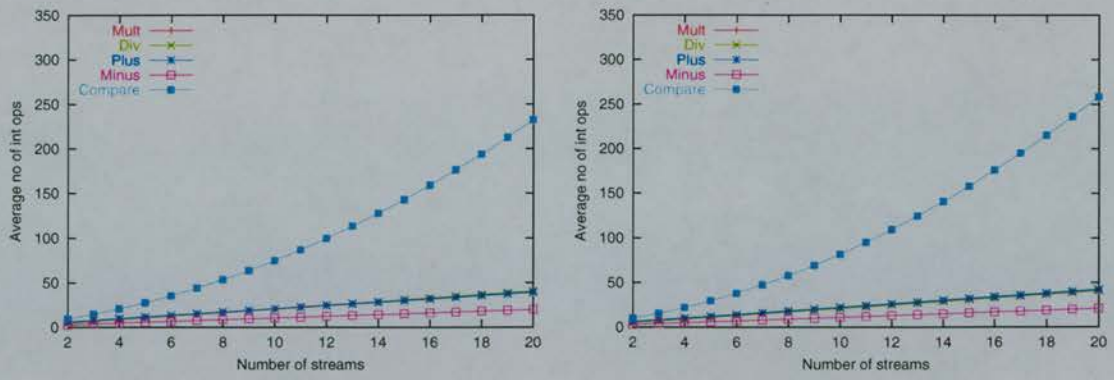
Like burst creation, the accuracy of burst splitting is related to the accuracy of the burst ACTT value. In the case where the ACTT multiplier is 1000, a burst is a good representation of the multiplexed cell streams it describes, due to the accuracy of the burst ACTT. When the burst is split, the likelihood of an overestimate of the split time period is small, and the technique will not have

to apply cell count scaling for overestimated component streams. The drawback of mostly having first allocations less than the required number of cells (to fill the split time period) is that the code for cell addition based on cell fractions has to be run nearly every time. When the multiplier is 10, it is clear that the number of cells to consider, coupled with the number of streams, has a major impact on the success of the first allocation of cells (as shown in Figure 4.11). As the burst ACTT is an overestimate, the proportion of cells from each stream allocated to the split will also be an overestimate, hence the higher percentage of "Greater" bursts for the larger burst size experiments (Batch 6). This forces the technique to scale the component stream cell counts down, and then make up any shortfall produced by this technique nearly every time (as can be seen from the "< after scaling" line in Figure 4.11(b)).

Figure 4.12 shows the average integer instruction counts for the burst split experiments in batches 5 and 6. The average is taken over the number of bursts produced in each testbed program run. There is very little difference between the instruction averages for the cases where the ACTT multiplier is 1000 (Figures 4.12(b) and 4.12(d)), as would be expected due to nearly 100% of the split bursts produced being underestimates of the time period to fill. In this case, the code has to increment the cell counts for each component stream based on cell fractions contributing to the split burst. There is no such similarity in the case where the ACTT multiplier was 10 due to the differences in the success of the first allocation as seen in Figure 4.11. For the bursts with the lower size range (Batch 5), most of the splits produced are underestimates of the split period, so the addition by cell fractions code needs to be executed, thus producing similar operation averages to the case where the multiplier is 1000. When the cell count per burst is increased by an order of magnitude (Figure 4.12(c)) a larger proportion of split bursts are overestimates, so the cell counts need to be scaled. As can be seen from the line denoting the number of underestimates after the scaling process (Figure 4.11(b)), most split bursts then have to have their cell counts incremented by addition of fractions. The extra workload caused by the initial inaccuracy of the burst ACTT, coupled with the large number of cells and streams, leads to the different mix of instructions as shown in Figure 4.12(c).

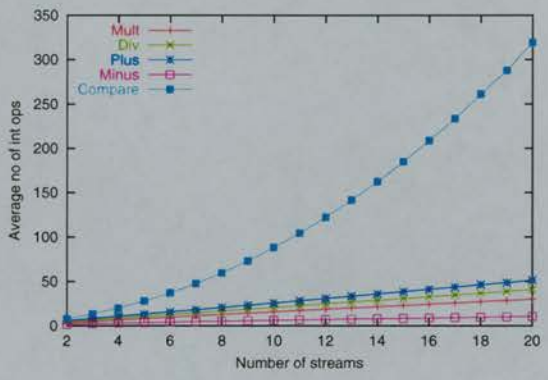
4.3.4 Summary

Like burst creation, the accuracy of the burst ACTT value, as well as the number of cells carried in each burst, have a bearing on the performance of burst splitting. For an accurate burst ACTT (due to using a large ACTT multiplier), the

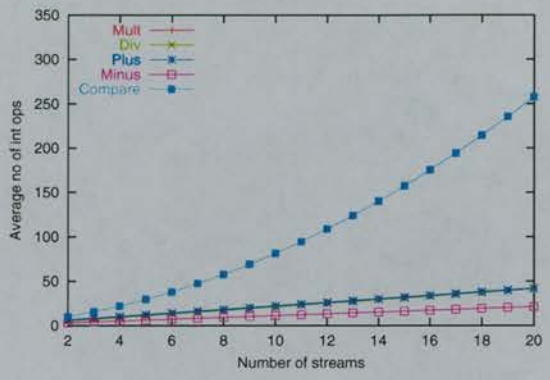


(a) ACTT=10, cells=(10,1000)

(b) ACTT=1000, cells=(10,1000)



(c) ACTT=10, cells=(100,10000)



(d) ACTT=1000, cells=(100,10000)

Figure 4.12: Average number of integer instructions per split burst produced in experiment batches 5 and 6

total number of cells allocated at the first allocation of cells from the component streams mainly underestimates the number of cells required to fill the split time period. The shortfall means that the technique has to add cells from component streams based on contribution fractions. For a less accurate burst ACTT (due to using a low ACTT multiplier), the likelihood that the total number of cells produced at the first allocation overestimates the number required, increases with increasing burst size and component stream count. Having to scale the overestimated stream cell counts, before perhaps having to add cells by fractions to reach the cell count required to fill the split time, increases the work necessary in the burst splitting procedure.

4.4 Improving integer performance

4.4.1 Introduction

The previous sections examined the accuracy and performance issues relating to two core techniques of the integer burst-level simulation technique. Not only does the choice of precision of the variables used in the techniques have a strong influence on their final accuracy, but also on the computational complexity of each technique. Reducing the computational cost of the core techniques is one way of ensuring the entire simulation environment is as efficient as possible. One of the reasons for choosing an all-integer approach was to maximise the performance advantage of integer arithmetic in modern microprocessors (as the performance of integer arithmetic instructions will always be better than that of the floating point instructions). Optimising the integer arithmetic performed will yield even higher gains in performance. It is important, however, to ensure that applying further optimisation to an already abstracted technique does not greatly reduce the accuracy attainable by such a technique.

4.4.2 The cost of integer instructions

As was shown in Table 3.1 on page 32, integer arithmetic instructions in modern microprocessor cores have lower latencies and throughputs than the equivalent floating point operations. It is clear that division of either integer or floating point operands is the most costly instruction type in terms of latency and throughput. The situation is worse when considering the use of 64-bit variables on a 32-bit architecture as the intrinsic arithmetic operations are usually performed by subroutines defined by the compiler used. This is necessary to perform 64-bit arithmetic on the Pentium II or K6 but it will introduce an even higher overhead. The use of a true 64-bit architecture such as the Compaq *Alpha*[51] or Intel *Itanium*[52] would provide native 64-bit arithmetic operations as standard. However, reducing the complexity of key algorithms at the source code level, and removing as many divide instructions as possible, will have the best chance of producing code with the best performance after compilation. Accurately predicting performance from the assembly language produced by a compiler is difficult as modern microprocessors will use techniques such as speculative execution, out-of-order processing (dynamic scheduling) and multi-stage pipelines in order to hide the latency of instructions issued. For example, if a piece of code contains many simple integer operations (such as add and multiply) and few costly floating-point operations (such as multiply and divide) the cost of issuing the FP instruction may be amor-

tized if the processor can schedule enough integer instructions to keep busy. If enough out-of-order instructions can be issued, there may be no wait penalty before the result of the FP instruction is actually required for further computation.

The high cost of division based on the standard *subtract and test* algorithm has been known for some time, as this algorithm is difficult to pipeline. For example, the CRAY-1[59] supercomputer used a Newton-Raphson iterative algorithm to calculate the reciprocal of a floating point divisor before performing a subsequent multiply to produce the result of the division. Such techniques are now used in the enhanced instruction sets of recent microprocessors for faster floating point division (eg. the *3DNOW!*[4] instructions in all new AMD microprocessors).

4.4.3 Integer division by reciprocal multiplication

As shown in Table 3.1, division, be it integer or floating point, is potentially a costly instruction to issue when one wishes to produce fast and efficient algorithms. If an algorithm relies on issuing several divide instructions in short succession, the potential for the execution stalling is high and this could result in a drop in the performance achievable. However, it is possible to apply an approximation to improve the performance of integer division and indeed this is a recommended optimisation from the processor manufacturers (see [4] and [62]). The recommendation is to replace integer division with multiplication by the reciprocal of the divisor. The idea is to take the reciprocal of the divisor and multiply it by a large integer constant to make the reciprocal an integer. The dividend is then multiplied by the reciprocal and the product is divided by the original integer constant to give the result of the division. The first step, that of multiplying the reciprocal of the divisor by an integer constant is equivalent to dividing the integer constant by the divisor. Figure 4.13 shows the C code for a sequence of integer divisions each using the same divisor, and that of the version replacing each division by multiplication of the reciprocal.

Figure 4.13 shows how instructions with lower latencies can be used to perform the sequence of integer divisions. Clearly, the reciprocal multiplied by the integer constant (`SHIFT`) has to be computed and this is an integer division. In the case where only one division has to be performed this would increase, rather than decrease, the overall latency but a performance benefit is obtainable when many divisions are performed using the same divisor as the reciprocal need only be calculated once. The example also shows that if the integer constant used to produce the reciprocal is a power of 2 then shift instructions can be used which have very low latencies (as shown in Table 3.1). Left shifting (`<<`) a number by a

```
int i, arg1[5], arg2;
int result[5];
...
for(i = 0; i < 5; ++i){
result = arg1[i] / arg2;
}
```

(a) Original C code

```
int i, arg1[5], arg2;
int result[5];
int SHIFT, lsr;
SHIFT = 1 << 16;
...
lsr = SHIFT / arg2;
for(i = 0; i < 5; ++i){
result[i] = (arg1[i] * lsr) >> 16;
}
```

(b) Replacing the divide

Figure 4.13: Replacing integer division with multiplication by the reciprocal of the divisor

power of two is equivalent to multiplying that number by 2 raised to that power, ie. $a<<x = a \times 2^x$. Right shifting (>>) a number by a power of two is equivalent to dividing that number by 2 raised to that power, ie. $a>>x = a/2^x$.

4.4.4 Pitfalls of integer division by multiplication of reciprocals

Replacing integer division by multiplication by the reciprocal of the divisor is an approximation, as an integer value is used to represent a floating point quantity. Floating point variables within a computer are also an approximation of real numbers due to the limited storage space assigned to them. However, the integer approximation of the reciprocal value will depend on the magnitude of the shift power chosen as this affects the *granularity* of the range of values available. Choosing a large power of two for the integer constant will improve the accuracy of the result, but may potentially lead to integer arithmetic overflow at the stage where the left shifted reciprocal is multiplied by the dividend. Choosing too small a power of two for the integer constant will lead to poor accuracy of the result. If the entire range of values for the dividend and divisor are known, a suitable integer constant can be chosen to prevent integer overflow. The range of the arguments may also define the storage allocated to each variable as 64-bit variables will give much more leeway for choice compared to that offered by 32-bit variables. The drawback would be that intrinsic arithmetic for 64-bit variables is calculated with 32-bit arithmetic on 32-bit architectures, which requires extra overhead. However, the replacement of division by low latency multiply and shift instructions should still yield a benefit even for 64-bit operations on a 32-bit

machine.

Another effect of the granularity in the range of the left shifted reciprocal is that, when the dividend is slightly larger than an integral number of divisors, the answer returned could be one less than it should be. This is because the left shifted reciprocal is effectively an underestimate of the true value as the integer divide used to calculate it returns the integer floor of the result. Thus when the left shifted reciprocal is multiplied by the dividend, the product will be slightly less than the integer constant multiplied by the true answer. When the product is right shifted to produce the answer, the effect of the underestimate is to return an answer one less than the true value. A solution to this is to increment the value of the left shifted reciprocal by one such that it slightly overestimates the true value. This has the effect of returning *at worst* the integer ceiling of the division depending on the granularity of the left shifted reciprocal values. Care has to be taken in the case where the dividend is less than the divisor, as this case can either return the result zero or one depending on the requirement for the result. If the ceiling is required for every division, then the technique should return the answer 1 in this case. If, however, the floor of the result is required, the left shifted reciprocal will not need to be incremeted by one, and the division should return zero if the dividend is less than the divisor.

The steps for integer division by multiplication of left shifted reciprocals are as follows (when the ceiling of the result is required):

1. Choose a value for the integer constant
   (eg. `int SHIFT = 1 << 16;`)

2. Is the dividend greater than the divisor? If not return one straight away as the answer
   (eg. `if(dividend < divisor) return 1; `)

3. Else produce the left shifted reciprocal
   (eg. `int lsr = (SHIFT / divisor) + 1;`)

4. Multiply the left shifted reciprocal by the dividend
   (eg. `int product = lsr * dividend;`)

5. Right shift the product by the power of the integer constant to produce the answer
   (eg. `int answer = product >> 16; return answer;`)

To illustrate the technique, a simple test was performed where 1000000 standard 64-bit integer divisions were performed and tested against the results from

64-bit division by multiplication of reciprocals. The power of 2 for the integer constant was chosen for each run and the dividend and divisor were drawn from a random distribution of integers in the range 1 to 1000000 (ie. $Randint(1, 1000000)$). The absolute values of each division were compared as well as the results of producing the integer ceiling of the result of the division. The integer ceiling is the highest integral value not less than the result of the divison. If $y = a/b$ then y is incremented by 1 if $a > (y \times b)$. Figure 4.14 shows the percentage of result differences when the left shifted reciprocal was the standard result of the reciprocal calculation (ie. $\lfloor \texttt{SHIFT}/divisor \rfloor$) and when it was incremented by 1 (ie. $\lfloor \texttt{SHIFT}/divisor \rfloor + 1$).

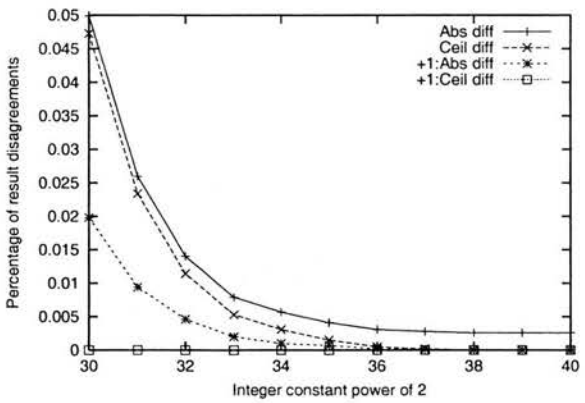


Figure 4.14: Percentage of disagreements between standard integer division and division by multiplication by reciprocals for normal and incremented left shifted reciprocals

Figure 4.14 confirms that the accuracy of division by multiplication of reciprocals improves when the size of the integer constant is increased. It also shows that the number of differences between standard integer division and the new technique are reduced when the left shifted reciprocal is incremented by 1 (the lines marked "+1:Abs diff" and "+1:Ceil diff" in the figure). In the case of the integer ceiling result, the new technique reports no differences for the parameters used in the test. The case where the standard left shifted reciprocal is used (lines "Abs diff" and "Ceil diff" in the figure) show that the loss of accuracy produces more disagreements. Further investigation showed that many of the disagreements in this case were due to the dividend being slightly greater than some integral number of divisors in each instance.

4.5 Revising the core techniques

4.5.1 Introduction

As demonstrated in Section 4.2 and Section 4.3, the core techniques of burst creation and to a lesser extent burst splitting, can require a large number of integer divide operations to be executed. Standard integer division can be a costly operation in modern microprocessors (as described in Section 4.4.2), but its impact can be reduced by replacing integer division by multiplication of reciprocals. As this technique still requires an integer division operation to produce the left shifted reciprocal in the first place, a performance benefit will only be realised if several integer divides can be replaced using each left shifted reciprocal calculated. Analysis of the two core techniques shows that many divisions are performed using the burst ACTT as the divisor. If the left shifted reciprocal of each ACTT is calculated when a burst is created, all subsequent calculations requiring a division by that ACTT can be replaced with division by multiplication of the reciprocal.

4.5.2 A first look at replacing integer division in burst ACTT calculation

It is clear from the performance data that burst ACTT calculation is very costly in terms of the number of integer divides and other operations required to produce the result. This is due to having to find the least common multiple covering all of the component stream ACTTs in the multiplexed burst. This operation in turn requires many calculations to find greatest common divisors of pairs of numbers. However, it is possible to replace this computationally expensive process with a simpler method based on the left shifted reciprocals calculated for each ACTT. The key to this simplification is due to the fact that the ACTT is the *inverse* of the *cell arrival rate* for the cell stream.

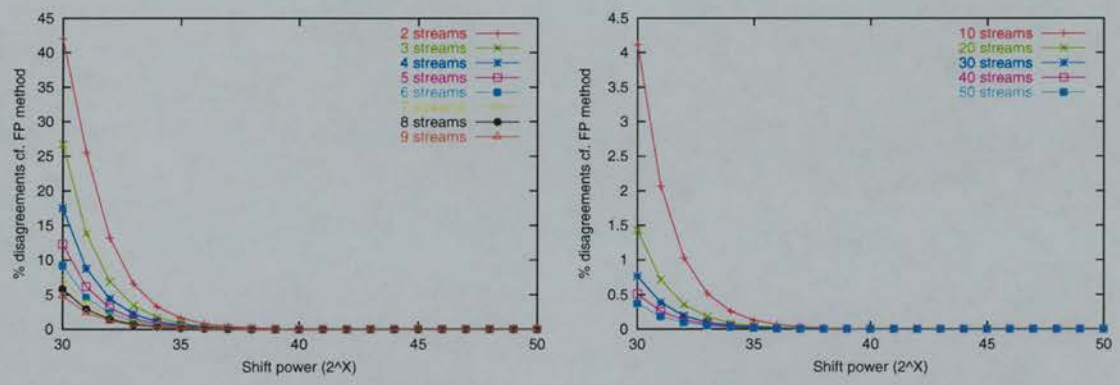
Thus, if $ACTT_B$ is the burst ACTT of a multiplexed burst containing N component streams each with an ACTT of $ACTT_i$ then $ACTT_B$ can be calculated by Equation 4.5.

$$\frac{1.0}{ACTT_B} = \sum_{i=1}^{N} \frac{1.0}{ACTT_i} \tag{4.5}$$

If each cell stream has a left shifted ACTT reciprocal of lsr_i and an integer constant `SHIFT` is used in the calculation, the ACTT for the burst ($ACTT_B$) can be calculated by Equation 4.6.

$$ACTT_B = \frac{\texttt{SHIFT}}{\sum lsr_i} \tag{4.6}$$

To examine the effect of using the new integer method for finding the burst ACTT, a simple test program was created to compare the burst ACTT value produced by the new method against that produced by the simple floating point method (as shown in Equation 4.1). Each run of the test program allowed for the integer shift factor to be changed, as well as the number of input streams to be considered in each burst creation. The stream ACTT values were chosen each time from a random distribution in the range 10 to 100000 (ie. $Randint(10, 100000)$) and 1000000 bursts were produced in each run to give an average. The results of the percentage of burst ACTTs produced by the new technique which differed from the results of the floating point calculation are shown in Figure 4.15.



(a) 2 to 9 cell streams per burst

(b) 10 to 50 cell streams per burst

Figure 4.15: Percentage of differences between the new integer burst ACTT calculation and the floating point method for varying cell streams and integer constant values.

As can be seen from Figure 4.15, the number of disagreements decreases with both increasing integer constant and increasing the number of streams in each burst. This is as expected, as the larger the integer constant, the better the accuracy of the left shifted reciprocal. Increasing the number of component streams lowers the value of the burst ACTT calculated, making it more likely that the two techniques will produce the same integer ceiling result. Examination of the absolute difference between the burst ACTT values returned from the new integer method and the floating point technique revealed that in nearly 100% of cases the difference was 1. There was a tiny amount of variability in the absolute difference for small numbers of streams (less than 3) when the integer constant was small (less than 2^{32}), but the absolute difference was mainly of value 1. Depending on

the relative magnitudes of the ACTT values considered, an absolute difference of 1 from the ideal is tolerable. However, it is worth noting that the least common multiple integer technique for finding the burst ACTT recorded no differences in the value, when compared to the ideal, in the experiments in batches 1 to 4.

Apart from producing burst ACTT values relatively close to the ideal value in each case, the new technique bypasses the potential fragility of the the least common multiple based integer technique. As described in Section 3.4.2, it is possible for integer overflow to occur when the stream ACTTs considered for a burst have high valued common multiples. This can lead to very large values of the least common multiple, causing overflow even in a 64-bit variable. The new technique does not have the same limitation, but is limited in the situation where the sum of the stream left shifted reciprocals approaches the value of the shift constant chosen. A suitable large factor will allow for many cell streams to be multiplexed without the danger of integer overflow. The tradeoff for this advantage is that of accuracy as the left shifted reciprocal accuracy improves with increasing integer shift constant, but the larger the integer shift constant the greater the likelihood of overflow in integer division calculations. The integer constant also has to be large, relative to the maximum ACTT value, as otherwise this will limit the accuracy of ACTT values. A large constant will give increasingly better left shifted reciprocal accuracies with decreasing maximum ACTT size.

Before a simulation study commences, the modeller will have to determine suitable sizes for the integer shift constant and the ACTT multiplier used (which increases the range and accuracy of burst ACTT values). Choosing suitable constants will involve assessing the range of likely ACTT values which will exist in a simulation run, along with likely burst sizes and the degree of the multiplexing which will be encountered. Large burst sizes coupled with a large ACTT multiplier would influence the accuracy of integer division operations, thus requiring careful selection of the integer shift constant. A high degree of multiplexing will require a large ACTT multiplier to ensure accuracy in burst ACTT calculations. With these factors in mind, the modeller can choose a level of accuracy and then fit the integer shift constant and ACTT multiplier values around this.

With the various limitations in mind tradeoffs between accuracy, runtime performance and simulator scalability can be made through the choice of integer techniques selected. The choice of the target architecture for the simulator will also have an impact in the decision making process, as a native 64-bit architecture would deliver immediate benefits to each technique. Considering that integer shift constants greater than 2^{32} have been shown to be important for ac-

curacy purposes, 64-bit arithmetic is essential to the improved integer technique for calculating the burst ACTT.

4.5.3 Revising burst creation

The burst creation process as described in Section 4.2 suffers from the computational complexity and cost of the technique used for burst ACTT calculation. This makes burst creation a prime candidate for the techniques described in Sections 4.4.3 and 4.5.2 as a means of improving the efficiency. To investigate this, the simple testbed program used in Section 4.2.2 was modified such that the burst ACTT calculation used the new integer technique described, and that all possible integer divides by ACTT values were replaced by multiplication of reciprocals. The left shifted reciprocal of each stream ACTT was calculated after the stream ACTT values were chosen, and this integer divide was counted in the instruction averages. Each left shifted reciprocal was incremented by one, with this addition also counted in the instruction averages. Extra tests were added to the code so that any potential integer overflows from the new technique could be detected.

The first choice to make in the revised testbed program was that of the integer constant to be used for calculating the left shifted ACTT reciprocals. The testbed used 64-bit variables allowing both a large upper limit on the integer constant and flexibility in the ACTT values used. To help prevent overflow, an integer constant of 2^{36} was chosen, as this produces left shifted reciprocals in the range 2^{17} to 2^{30} for the chosen experimental parameters (ie. when the ACTT multiplier is 1000 and the ACTT value is 1000 and when the ACTT multiplier is 10 and the ACTT value is 10 respectively). This leaves, at worst, an upper bound of 2^{34} for the dividend. The dividend is typically a burst duration, so assuming the ACTT is 1000000 (ie. the worst case value when the multiplier is 1000 and the ACTT value is 1000), the cell stream can describe around 20000 cells in total before integer overflow is a potential problem.

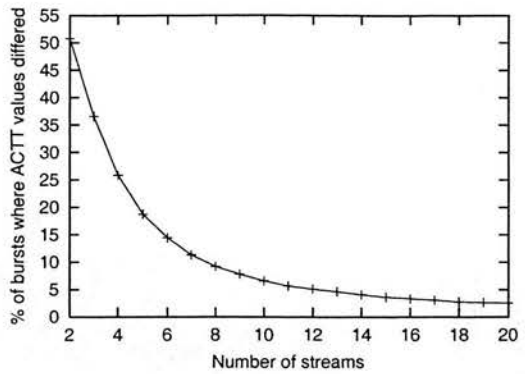
Experiments were conducted, as batches 7 and 8, with the revised testbed to mirror the experiments completed for the previous burst creation analysis. This would allow for direct comparisons to be made between the results for both techniques used. The values shown in Table 4.6 on page 93 were the parameters used in the experiments for batches 7 and 8. The results from the revised testbed runs are described in the following sections.

4.5.3.1 Accuracy comparison

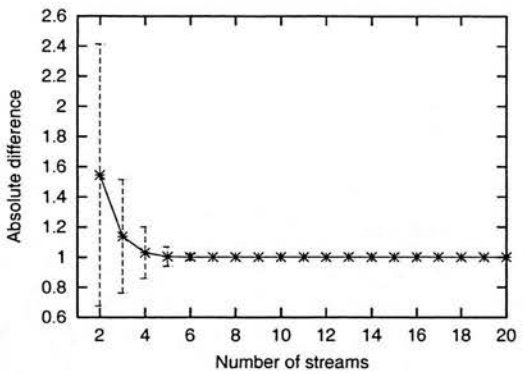
The first test of the revised testbed program was to see how the new burst ACTT calculation technique affected the burst ACTT values produced. When the ACTT multiplier was 10, there were very few disagreements between the integer ACTT calculation and the floating point equivalent with the percentage of differences being only a small fraction of a percent of the number of bursts produced. This was as expected due to the accuracy of the left shifted reciprocals increasing with decreasing maximum ACTT size. However, when the ACTT multiplier was 1000, the results were very different and they are shown in Figure 4.16. The results show what increasing the maximum ACTT size by two orders of magnitude for the same integer constant can do for the accuracy of the technique. However, the average value of the absolute difference decreased (converging to 1) with increasing numbers of component streams. Having an absolute difference greater than or equal to 1 can be tolerable when the absolute size of the ACTT is relatively much larger making the error a very small percentage of the ACTT. Any error is magnified by the number of cells described by the burst so minimising the absolute error is still important. Choosing a larger integer constant would decrease the number of disagreements but could also lead to integer overflow problems in division by reciprocal multiplication operations. Analysing the possible range of ACTT values and burst sizes which will be encountered in a simulation run will allow an experimenter to choose an optimal ACTT multiplier. The choice of an optimal value will minimise the absolute error in the burst ACTT values calculated.

As calculating the burst ACTT does not rely on the number of cells in each component stream, the number and range of the differences were the same when the cell stream counts were chosen from the range 100-10000.

The next test compared the results of the first allocation of cells to the multiplexed burst based on the computed time period for the burst. As the relative difference in burst ACTT values between the experiments using the different techniques was small, the first allocation of cells was virtually identical between the results from each technique. Hence, the considerations of ACTT size, ACTT accuracy and burst size for the results of batches 3 and 4 (see Section 4.2.6.1) are equally applicable when the new integer techniques were used. The only difference when using the new integer techniques was that an extra check had to be introduced after the first cell allocation to ensure that overestimates were trapped. This extra check basically compared the time period described by the number of cells allocated, and if this was greater than the time period to be filled



(a) % of bursts with ACTT differences



(b) Average (with sd) of absolute difference

Figure 4.16: Percentage and absolute difference between integer and FP calculated burst ACTT values when the ACTT multiplier is 1000 and the cell ranges are 10-1000 when new integer technique is used.

then the number of cells was decremented by one. The integer divide by reciprocal multiplication technique was tuned to return *at worst* the integer ceiling of the division, but an overestimate in the cell allocation phase could be problematic. A symptom of not including the extra test was apparent at the addition of cells based on fractions stage. Integer overflows could occur as the "fraction" for a stream was meaningless due to over-allocation at the first stage producing erroneous counts.

The ultimate test for the new integer techniques used in burst creation was their effect on the final ACTT ratios (as described in Section 4.2.3). As in experimental batches 3 and 4, final ACTT ratios were collected for experiments where the number of component streams, distribution of cells per stream and the ACTT multiplier used could be chosen. The same parameters were used for the experiments as were used for batches 3 and 4 (see Table 4.6 on page 93). A first look at the results from the experiments revealed that the ACTT ratios were almost identical to the results for the same parameters in batches 3 and 4. To test this, a linear regression test was made, plotting the average ACTT ratio from each experiment in batches 7 and 8 against the average ACTT ratio from the experiment with the same parameters in batches 3 and 4 (ie. comparing like for like experiments with the only difference being the choice of integer techniques). A data point consisted of the following:

x-coordinate the value of the average ACTT ratio for some number of component streams using the old integer techniques.

y-coordinate the average ACTT ratio for the same number of component streams (with all of the other parameters also being equal) when the new integer techniques were used.

The same approach was taken for the standard deviation of each average ACTT ratio. The results are shown in Table 4.8 with each experimental pairing (based on ACTT multiplier and distribution of cells to each component stream) on a separate row. The gradient of the line produced from the linear regression is shown along with the correlation coefficient (R^2) which gives a measure of the goodness of fit of the correlation line to the data. The closer the correlation coefficient is to 1, the better the fit of the line produced by linear regression to the spread of the data.

Experiment		Average ACTTratio		Standard deviation	
ACTT mult	Burst size	Gradient	R^2	Gradient	R^2
10	10-1000	1.00000	1.000000	1.000000	1.0
1000	10-1000	1.00042	0.999999	0.999941	1.0
10	100-10000	1.00000	1.000000	1.000000	1.0
1000	100-10000	1.00119	0.999129	0.999934	1.0

Table 4.8: Linear regression results of comparing average ACTT ratio results between the experiments using division by reciprocal multiplication compared with those using standard integer division.

As can be seen from Table 4.8 the new integer techniques had a very slight impact on the accuracy of burst creation when compared to the results using the original integer techniques. The results for an ACTT multiplier of 10 are virtually identical in either case, and this is as expected due to the accuracy of the left shifted reciprocals for small magnitude ACTT values. There is some difference when the multiplier is 1000 due to the greater chance of some inaccuracy in the burst ACTT value due to the new integer technique. The overall effect, for the experimental parameters chosen, is that the combination of new integer techniques for large ACTT values slightly *overestimates* the average ACTT ratio which is shown by a gradient greater than 1 (a gradient of 1.0 denotes perfect agreement between the results). The overestimate increases slightly when a greater number of cells are considered in each burst. This is as expected due to the inaccuracy in the initial burst ACTT being magnified by the number of cells under consideration. The integer techniques are sensitive to the parameters used to calculate each burst ACTT. The new integer technique, especially, is less accurate for increasing component ACTT sizes when a relatively small integer shift constant is used.

The final point of note is that no potential integer overflow operations were recorded in any of the experiments confirming that the integer constant used was at least large enough for the experimental parameters chosen. Any potential overflow could be caught by checking the size of the left shifted reciprocal and that of the dividend to ensure that their product can fit in 64-bits. As the range of left shifted reciprocals is known, the initial test for potential integer overflow is whether the dividend is larger than the largest possible value which can be safely multiplied by the largest left shifted reciprocal. The relative magnitudes of both arguments can then be determined if this test is triggered and if the product will overflow 64-bits, the larger of the two arguments can be right shifted to prevent overflow. To produce the final answer of the intended divide instruction, the product of the dividend and left shifted reciprocal is then right shifted by a power suitably reduced to take account of the previous right shift of one of the arguments. The power used in the first right shift is the minimum possible to allow the product to fit in 64 bits. This will, of course, lower the accuracy of the result returned. However, if the value of the integer shift constant is chosen carefully (after examining the ranges of ACTT values in a simulation), the cases where a potential integer overflow could occur can be minimised to be rare events. If the approximation to prevent integer overflow is used infrequently in a simulation run, the accuracy of the techniques will be scarcely affected.

4.5.3.2 Performance factors

The results presented in the previous section detailed the very small impact of the new integer techniques on the accuracy of burst creation when compared with the results from using the previous techniques. The prime motivation for testing the new techniques was their likely impact on the performance factors for burst creation. Removing integer divides, which are the most costly instructions in terms of latency, should have a good effect on the runtime performance of the algorithms used in burst creation. To gain a feel for the performance factors due to the new techniques, the experiments performed in batches 7 and 8 were annotated to count the number and type of integer operations executed in the same way as the experiments in batches 3 and 4 were. The instruction counts were averaged over the number of bursts produced in each run with specific average breakdowns reported for the overall operation of the technique, the calculation of burst ACTTs and the addition of cells based on fractions.

As anticipated, the adoption of the new integer techniques dramatically reduced the number of integer divide operations required in the experiments per-

formed. Rather than present individual graphs of how the instruction mix varied with the number of cell streams, Table 4.9 presents the average number of integer instructions executed when 20 individual cell streams were merged to produce each burst. The data comes from both the experiments conducted in batches 3 and 4 (denoted "Old" in the "Int Div" column) and the experiments conducted in batches 7 and 8 (denoted "New"). The choice of the experiments for 20 cell streams was made to highlight the average differences for the experiments requiring the most calculations.

Experiment			Integer operation averages for 20 streams					
Int Div	ACTT mult	Burst Size	×	÷	+	−	Compare	Shift
Old	10	10-1000	94.6	191.7	33.5	131.5	266.7	n/a
New	10	10-1000	111.9	22.0	75.4	17.1	176.1	37.3
Old	1000	10-1000	99.9	227.5	41.1	167.5	380.3	n/a
New	1000	10-1000	119.9	22.0	83.1	20.0	253.8	39.9
Old	10	100-10000	70.8	179.5	24.2	118.7	148.3	n/a
New	10	100-10000	75.9	22.0	66.1	4.2	69.8	25.0
Old	1000	100-10000	99.9	227.5	40.2	167.5	371.5	n/a
New	1000	100-10000	119.9	22.0	82.2	20.0	245.4	39.9

Table 4.9: Comparing the integer operation averages for the experiments where 20 component cell streams are merged (to produce a burst) for standard integer division and division my reciprocal multiplication.

It is immediately clear from Table 4.9 that the desired effect of reducing integer divide operations is satisfied by the integration of the new integer techniques into burst creation. The average number of divide operations is reduced to virtually a constant which depends on the number of streams to merge for a burst. The divide operations reported in the new technique are used to determine the left shifted reciprocals for each ACTT, as well as in the final calculation to produce the burst ACTT. The number of subtract operations is also reduced, and this is mainly due to no longer requiring integer modulo arithmetic when calculating the burst ACTT. Integer multiplication and addition instruction average counts have increased, but their very low latencies, when compared with integer division, help mitigate this. The new techniques introduce the shift operation but this also has a very low latency. Another advantage is that the number of comparison operations is reduced when using the new technique (mainly due to replacing the complicated burst ACTT calculation) and this may help contribute to a runtime efficiency improvement as the host microprocessor will have less branches to predict and cope with when scheduling instructions.

4.5.3.3 Runtime implications

The results presented in the previous section show that the new integer burst creation technique reduces the number of integer divide operations necessary (when compared to the previous integer technique). However, the new technique results in an increase in integer instructions with lower latencies and throughputs. To see how the change in the average instruction mix affects runtime performance, the burst creation testbed program was altered to give the total runtimes for the experiments involving each of the integer techniques for burst creation. The testbed was stripped of all the annotation code so that the only work performed was that of burst creation using whichever integer technique was selected. The experiments in batches 3 and 4 were rerun with the modified testbed to give the runtime performance data for the original integer burst creation technique. To get the same data for the new integer technique, the experiments from batches 7 and 8 were also rerun with the modified testbed program.

Table 4.10 presents the results of the testbed program runtimes (in seconds) for each of the integer burst creation techniques. The experiments for batches 3 and 4, using the original integer technique, are in the columns marked "Old" in the table. The results for batches 7 and 8, using the improved integer technique, are in the column marked "New" in the table. Each experiment was performed on the same workstation, with effort made to ensure that it was unloaded by other user processes.

As can be seen from Table 4.10, the use of the improved integer technique leads to a reduction in the total runtimes of each of the burst creation experiments performed. Figure 4.17 presents the relative speedup of the improved integer technique over the original integer technique for each experiment performed. The improved integer technique offers a runtime speedup for virtually every experiment performed. However, the extra work necessary to compute the left shifted reciprocals means that the improved integer technique is actually slower than the original technique when only two cell streams are being multiplexed. The speedup improves with an increasing number of streams to multiplex, as would be expected.

The results presented in this section show that the improved integer burst creation technique offers a raw runtime performance benefit over the original integer technique proposed in Chapter 3. However, the runtimes are taken from a testbed program designed to exercise the burst creation technique, rather than from the full simulation environment. As the full simulator has the event list to manage, as well as having to perform the other operations necessary for each simulation

ACTT mult	10				1000			
Size	10-1000		100-10000		10-1000		100-10000	
Streams	Old	New	Old	New	Old	New	Old	New
2	2.12	2.16	1.95	1.99	2.22	2.27	2.21	2.27
3	2.63	2.57	2.42	2.37	2.71	2.69	2.71	2.67
4	3.14	2.92	2.88	2.68	3.24	3.00	3.23	3.00
5	3.59	3.36	3.30	3.09	3.69	3.45	3.69	3.46
6	4.18	3.72	3.84	3.41	4.30	3.83	4.28	3.82
7	4.71	4.10	4.30	3.73	4.82	4.18	4.81	4.20
8	5.19	4.41	4.77	4.01	5.35	4.53	5.34	4.53
9	5.67	4.76	5.24	4.32	5.90	4.88	5.90	4.88
10	6.20	5.15	5.67	4.69	6.49	5.31	6.49	5.30
11	6.70	5.53	6.13	5.03	7.09	5.70	7.09	5.68
12	7.25	5.88	6.64	5.32	7.73	6.10	7.71	6.06
13	7.79	6.25	7.13	5.63	8.34	6.46	8.34	6.45
14	8.37	6.54	7.66	5.93	9.00	6.80	9.00	6.74
15	8.91	6.90	8.14	6.27	9.64	7.17	9.63	7.14
16	9.46	7.28	8.63	6.59	10.26	7.53	10.26	7.52
17	10.00	7.62	9.12	6.87	10.89	7.92	10.87	7.91
18	10.46	7.95	9.57	7.15	11.43	8.29	11.45	8.24
19	10.99	8.30	10.07	7.49	12.04	8.67	12.03	8.62
20	11.55	8.85	10.55	7.76	12.69	9.06	12.67	9.01

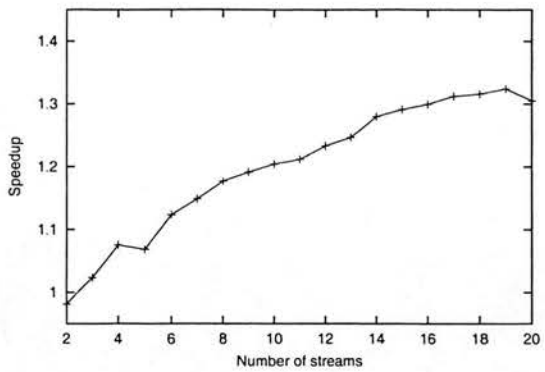
Table 4.10: Total runtimes (in seconds) for each experiment using the original and improved integer burst creation techniques

object, the choice of burst creation algorithm is not the only factor in determining the runtime performance. However, ensuring that each core technique is as fast as possible will be of benefit when the techniques are integrated into the simulation objects in the full simulator.

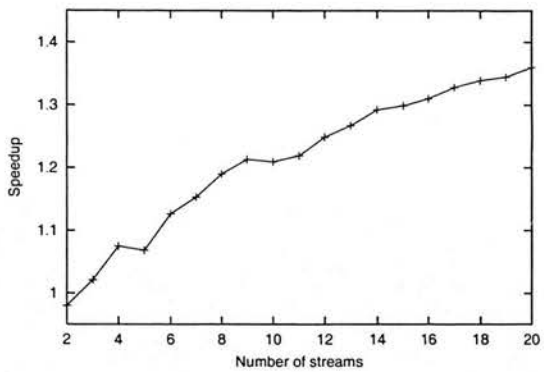
4.5.4 Revising burst splitting

Unlike burst creation, the technique of burst splitting has relatively little to gain from the new integer techniques. The major improvement in burst creation was due to the replacement of the complicated burst ACTT calculation by a simpler and more efficient technique. Burst splitting works with bursts which have already been created and thus have a burst ACTT and cell counts for each stream described. However, burst splitting does rely on the accuracy of the burst ACTT for the work it has to do and does use integer divide instructions in its operation. The integer divide instructions can be replaced with multiplication by reciprocals, and this will have a bearing on the accuracy and performance of the burst splitting technique.

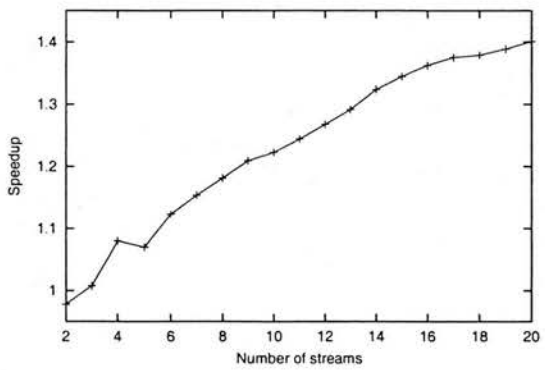
To test the effect of the new techniques on burst splitting, the testbed pro-



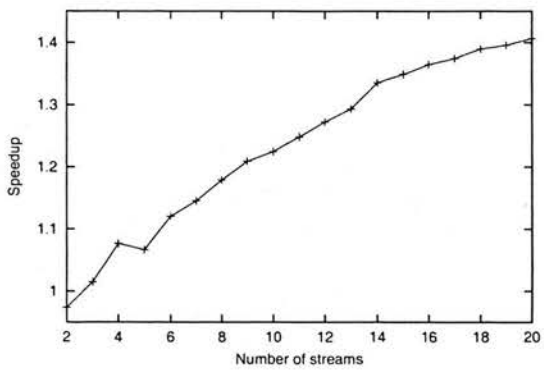
(a) ACTT=10, cells=(10,1000)



(b) ACTT=10, cells=(100,10000)



(c) ACTT=1000, cells=(10,1000)



(d) ACTT=1000, cells=(100,10000)

Figure 4.17: The relative runtime speedups recorded for the burst creation testbed when the improved integer techniques were used.

gram used in Section 4.3.2 was modified to enable division by reciprocal multiplication in the burst splitting routine. The same experiments were conducted as for batches 5 and 6 (see Table 4.7 on page 102) which analysed burst splitting with the original integer techniques. The new experiments, using the same parameters, were batches 9 and 10. In addition to logging how well burst splitting could fill the required time period at the first allocation and counting the instruction types, a log was made of the final burst split statistics. This was to allow comparison with the burst splitting results from experimental batches 5 and 6. Checks were also made for potential integer overflows.

4.5.4.1 Accuracy comparison and performance factors

The results from burst splitting, when the integer divides were replaced by division by reciprocal multiplication, were almost identical to the results from burst

splitting using standard integer division. This is pretty much as expected, with factors such as the accuracy of the burst ACTT (based on the ACTT multiplier used) having the most influence on the first allocation of cells just like it did for batches 5 and 6. The final split allocation of cells recorded in experimental batches 9 and 10 were identical to the allocations in the experiments with the same parameters in batches 5 and 6 in near 100% of the total number of split bursts produced.

In terms of the performance factors when division by reciprocal multiplication was used, Table 4.11 presents the comparative integer operation averages between the experiments in batches 5 and 6 (marked "Old" in the "Int Div" column) and the experiments in batches 9 and 10 (marked "New"). As expected, the integer divides used in the previous experiments have nearly all been replaced by an integer multiply and an integer shift operation in the new technique. The only integer divides remaining in the revised technique are used to find a left shifted reciprocal needed to scale down overestimated first allocations of cells. The latency penalty of the multiply and shift operations introduced is very much smaller than the integer divide latency and thus helpful in improving performance of the technique.

Experiment			Integer operation averages for 20 streams					
Int Div	ACTT mult	Burst Size	×	÷	+	−	Compare	Shift
Old	10	10-1000	39.2	40.4	39.2	19.7	232.2	n/a
New	10	10-1000	79.7	0.0	38.6	19.8	232.3	40.4
Old	1000	10-1000	40.4	40.7	41.7	20.7	258.1	n/a
New	1000	10-1000	81.4	0.0	41.6	20.7	278.3	40.7
Old	10	100-10000	29.7	40.5	51.1	10.2	319.0	n/a
New	10	100-10000	70.2	0.5	45.2	10.2	319.0	40.5
Old	1000	100-10000	40.9	40.9	41.5	20.9	257.2	n/a
New	1000	100-10000	81.9	0.0	41.5	20.9	277.2	40.9

Table 4.11: Comparing the integer operation averages for the experiments where bursts comprising of 20 component cell streams are split when standard integer division and division by reciprocal multiplication are used.

As with the revised burst creation experiments, there were no cases of potential integer overflow recorded in any of the experiments in the revised burst splitting experiments. Coupled with the equality of the final burst splits compared to the previous technique, this shows that the integer constant was at least large enough for the experimental parameters chosen.

4.5.4.2 Runtime implications

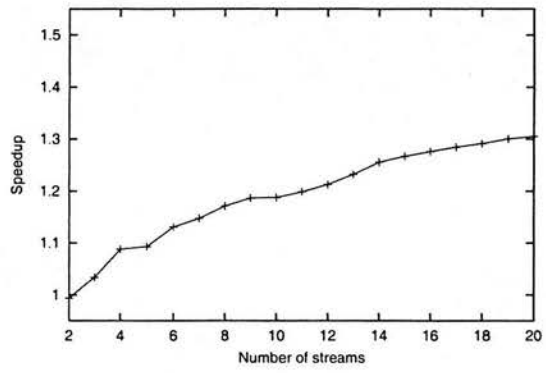
Unlike burst creation, the original burst splitting technique contains few integer divide instructions. However, these can be replaced with multiplication by reciprocals which results in an increase of instructions with lower latencies and throughputs than integer division. To assess the impact of the improved integer burst splitting technique on the runtime performance, the testbed program used for the burst splitting experiments was prepared to give total runtimes for each experiment. All of the annotation code was removed to leave just the burst splitting code. The experiments originally performed for the original burst splitting technique, batches 5 and 6, were rerun with the modified testbed program and the total runtimes were recorded. To get the same runtime data for the improved integer burst splitting technique, the experiments performed in batches 9 and 10 were rerun with the modified testbed program. All of the experiments were undertaken on the same workstation, with care taken to ensure that it was not loaded by other user processes.

Table 4.12 shows the runtimes (in seconds) for each of the experiments performed with the modified testbed program. The results in the "Old" columns are for the experiments which used the original integer burst splitting technique. The runtimes in the "New" columns are for the experiments which used the improved integer burst splitting technique.

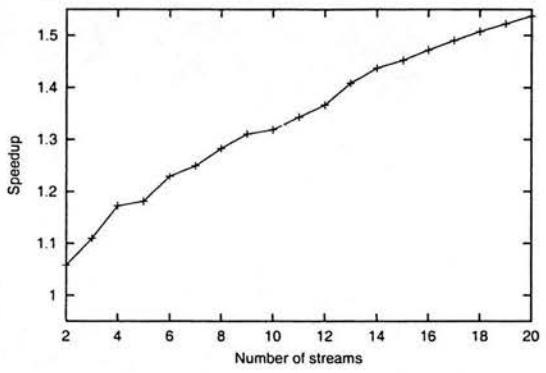
As can be seen from Table 4.12, the benefit of the improved integer technique for burst splitting is a reduced runtime. Figure 4.18 shows the relative speedup of the improved integer technique over the original for each of the experiments performed. The relative speedup due to the new integer burst splitting technique increases with an increasing number of streams to multiplex. This is as would be expected, due to the number of divides required increasing with the number of streams when using the original technique. Of interest are the speedups shown in Figure 4.18(b). In this case, the burst splitting procedure produces a large proportion of burst overestimates (see Section 4.3.3 on page 101). An overestimate of the split burst requires that each of the component cell stream counts is scaled, and then cells are added by fraction contribution as necessary. These operations increase the number of divide operations in the original technique, and so the improved integer technique offers an increased speedup by replacing these.

ACTT mult	10				1000			
Size	10-1000		100-10000		10-1000		100-10000	
Streams	Old	New	Old	New	Old	New	Old	New
2	3.32	3.34	3.46	3.27	3.46	3.49	3.45	3.46
3	3.96	3.83	4.04	3.64	4.04	3.93	4.03	3.94
4	4.58	4.21	4.69	4.00	4.69	4.35	4.67	4.35
5	5.17	4.73	5.28	4.47	5.28	4.89	5.29	4.88
6	5.86	5.18	6.00	4.88	6.00	5.35	6.00	5.35
7	6.45	5.62	6.61	5.29	6.61	5.81	6.59	5.80
8	7.04	6.01	7.25	5.65	7.25	6.22	7.26	6.23
9	7.63	6.43	7.89	6.02	7.89	6.65	7.90	6.65
10	8.22	6.92	8.56	6.49	8.56	7.18	8.57	7.18
11	8.81	7.35	9.27	6.90	9.27	7.68	9.27	7.66
12	9.52	7.85	10.03	7.34	10.03	8.15	10.04	8.14
13	10.18	8.26	10.79	7.66	10.79	8.62	10.79	8.62
14	10.80	8.60	11.53	8.02	11.53	9.00	11.51	9.00
15	11.48	9.06	12.26	8.44	12.26	9.50	12.26	9.49
16	12.11	9.49	13.00	8.83	13.00	9.97	13.00	10.00
17	12.76	9.93	13.74	9.22	13.74	10.49	13.75	10.44
18	13.42	10.39	14.52	9.63	14.52	10.96	14.49	10.97
19	14.10	10.84	15.27	10.03	15.27	11.46	15.26	11.45
20	14.75	11.30	16.05	10.44	16.05	12.00	16.05	11.96

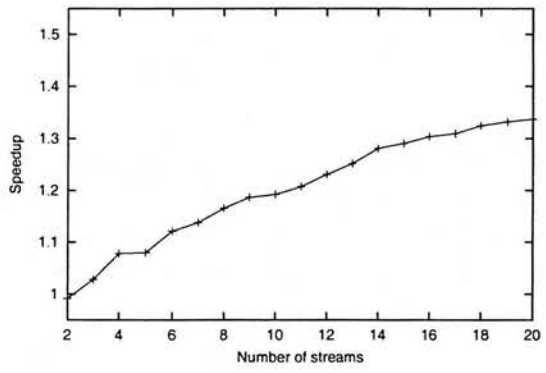
Table 4.12: Total runtimes (in seconds) for each experiment using the original and improved integer burst splitting techniques



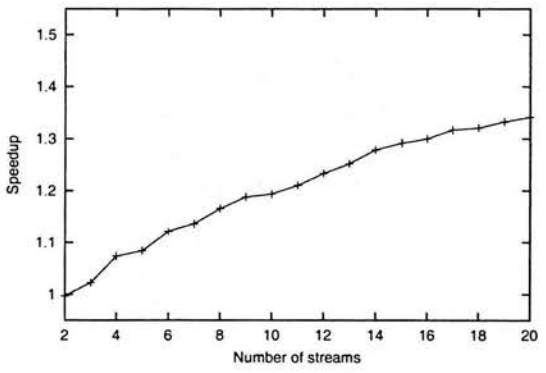
(a) ACTT=10, cells=(10,1000)



(b) ACTT=10, cells=(100,10000)



(c) ACTT=1000, cells=(10,1000)



(d) ACTT=1000, cells=(100,10000)

Figure 4.18: The relative runtime speedups recorded for the burst splitting testbed when the improved integer techniques were used.

4.6 Summary of chapter

The purpose of this chapter was to analyse in detail two of the core techniques used in the integer-time burst-level simulation environment proposed. The accuracy and the performance factors of each technique were examined.

The notion of an ACTT multiplier (used to increase the range of values possible for a burst ACTT) was shown to improve the accuracy of the techniques for large multiplier values. A minor drawback of a large ACTT multiplier was that each technique required the addition of cells by fractions (to ensure cell counts were correct) and this incurred a small performance penalty. However, a large ACTT multiplier was shown to help improve the accuracy of the techniques when many cell streams are considered in each burst. The accuracy benefits of a large ACTT multiplier are likely to outweigh the small cost of adding cell fractions in most cases.

The next part of the chapter looked at the relative execution costs of different integer arithmetic operations on modern microprocessors. The 32-bit integer divide operation was identified as having punitive latency and throughput costs, which would increase when considering 64-bit integer divide operations on a 32-bit architecture. In view of this, the method of replacing integer divide operations with multiplication by reciprocal operations was introduced. The idea is to replace each costly divide with simpler instructions with lower throughput and latency costs. Each of the core techniques was reworked to use this divide replacement, and the effects on accuracy and performance were presented. The improved integer techniques were shown to have a minimal impact on the accuracy of each technique, while delivering tangible runtime improvements.

It is clear that the use of the improved integer techniques is advantageous when many cell streams will be multiplexed at any one point in a simulation. Coupled to the use of a large ACTT multiplier, the improved techniques offer the best balance of performance and accuracy for the two core techniques presented.

Chapter 5

Simulator Performance

5.1 Introduction

The purpose of this chapter is to show how the burst-level simulation objects described in Chapter 3 perform when integrated into the simulation environment. Rather than attempting to cover the vast parameter space of simulator options – each of which will have some impact on performance – the emphasis is on attempting to quantify the effects of the detailed refinements made to the core techniques in Chapter 4, as well as the general options for the simulator presented in Chapter 3. To do this, the accuracy and runtime performance of the integer burst-level simulator are compared to an efficient cell-level simulator also produced in the work. Cell-level simulation is the most accurate way of modelling an ATM network, as well as being the most costly in terms of runtime performance. Comparing experiments performed in the burst-level simulator with identical experiments performed in the cell-level simulator, gives an ideal benchmark for performance and accuracy comparisons between the techniques. Identifying the circumstances under which the burst-level simulator performs well compared to the cell-level alternative, as well as the circumstances where it performs unfavourably, is a good means of assessing the technique.

The simple software architecture of the cell-level simulator is presented first, along with a description of the factors which hamper cell-level simulators. A summary of the parameter and behaviour options for the burst-level simulator is presented next to give a flavour of the number of combinations which will all influence the behaviour. Selected individual analysis of multiplexing, queueing and demultiplexing is then presented to illustrate the differences between the burst and cell-level simulators. Finally, the key findings and recommendations presented in the chapter are summarised.

5.2 A cell-level simulator

5.2.1 Introduction

It would be easy to produce a discrete event based cell-level simulator which would perform very badly when compared to the burst-level simulator designed in this work. However, by bearing in mind the efficiency improvements for discrete event simulators as described in Chapter 3, a good cell-level simulator can be produced. Of great importance is the event list management used in a cell-level simulator. This is because the number of events to process at any one time may be enormous, due to every cell "in flight" in the simulation being represented by an event. Hence, the use of an indexed event list or a post-ordered tree has the scope for greatly enhancing the performance realisable with the simple doubly-linked list variant. Both enhanced event list techniques are at the mercy of the distribution of event simulation times they are required to deal with, so choosing one in preference to the other is difficult. With the indexed event list in particular, the choice of the number of events to be represented by each event index can only be guessed at before commencing a simulation run. The choice of an optimal value, to reduce the runtime to a minimum, may only be possible after collecting the statistics available at the end of a previous run. Even if the modeller is satisfied with the discovery of such an "optimal" value, even a slight change in experimental parameters may alter the dynamics of a simulation run to the extent that the optimal value changes.

As with the burst-level simulator, there is a choice in how one models time in a cell-level simulator. A traditional floating point clock may be used, but there is no reason why an integer variable can not be used to store times. Translating a time represented by the integer clock is then a simple multiplication by some constant chosen by the modeller. As has been shown previously, integer arithmetic is preferable for performance reasons, and thus implementing the event list primitives with exclusively integer arithmetic will give a performance benefit for the cell-level simulator.

Another major design choice for a cell-level simulator is that of how one actually models cells by using discrete events. It would clearly be inefficient to have one event message to signal the start of a cell transmission and another to signal the end. This leaves the option of either placing the event either at the start, or at the end, of the cell transmission. Making the event signal the start of a cell would require that the cell carries information as regards its transmission duration. The side effect of this strategy is that each simulation object would have to perform

extra processing on each cell to ensure that the subsequent retransmission to other objects is correct. If the event marks the cell finish, this implies that the cell has completely arrived at its destination and may be transmitted elsewhere. No processing need be performed on the cell other than to ensure that the time gap between it and the last cell transmitted on the same link does not fall below the minimum for that link. Removing the need to store the transmission duration from a cell will reduce the memory footprint of a cell, thus making it possible to squeeze more cell events into the physical memory of the computer running the simulation.

Another significant design decision in a cell-level simulator is that of whether to buffer cells in simulation objects, or to directly schedule output cell events when a cell arrives at an object which uses buffering. The operation of a queue without flow control illustrates the difference between the strategies. If the modeller chooses to buffer cells in the queue object (which would be required if the cell arrival rate is larger than the cell output rate), the queue object will use more memory, but the number of pending events which it schedules will be small. This is because the queue can "sleep" between successive cell output transmissions, such that only one "wakeup" event need be pending at any one time to trigger the next output cell. Keeping the size of the event list as small as possible at all times is advantageous to the performance of the event list mechanism used. The other option is to directly schedule the output cell event for a cell as soon as its input cell event arrives at the queue. This is possible, as the queue can keep track of its "virtual occupancy" at any one time and the next simulation time at which it can produce an output cell event. When a cell arrives, the queue can decide how full it should be, and either discard the cell or schedule it at the next available output cell time which will be in the future. The advantage of this approach is that it requires no memory buffering in the simulation object and very little processing of the cells passing through. However, the number of cell events pending in the global event list at any one time may be very large, especially if many objects which utilise buffering are being modelled. The physical buffering option will actually increase the *total* event count for the simulation run (due to the need for "wakeup" messages to trigger the next output cell transmission), but may be faster due to the number of pending events in the event list being smaller, thus making the global event list processing faster.

With a myriad of options, the cell-level simulator is a research area in its own right. A great deal of research has been done on the possible ways of speeding up this simulation class through the use of parallel processing (as performed with

TimeWarp discrete event processing in Telesim[44]) or by redesigning the way the simulator handles cell transmissions (as in the CLASS[1] system). Each means of improving the cell-level simulator adds its own complications and possible assumptions. As the burst-level simulator produced in this work is an experimental research system, it is better to ascertain the potential of the technique by comparing it to the "standard" cell-level implementation, albeit made to be as efficient as possible. Comparing the raw performance of the burst-level simulator with that of alternative techniques is more attractive once the burst-level simulator has been proven in potential and then re-implemented for optimal performance.

5.2.2 The cell-level simulator

As with the burst-level simulator, the cell-level simulator produced for the work was implemented using a discrete event based system written in C++. The object structure of the simulator (Figure 5.1) is very similar to that of the burst-level simulator (as described in Chapter 3), as indeed they share the same indexed event and post-ordered tree event list structures. This is an advantage of using the object-oriented approach, as the changes required to implement the cell-level simulator were not dramatic. Another advantage is that any major runtime efficiency improvements, made in either the cell or burst-level simulators, may be shared by both systems to ensure that neither gets an unfair advantage when making comparisons.

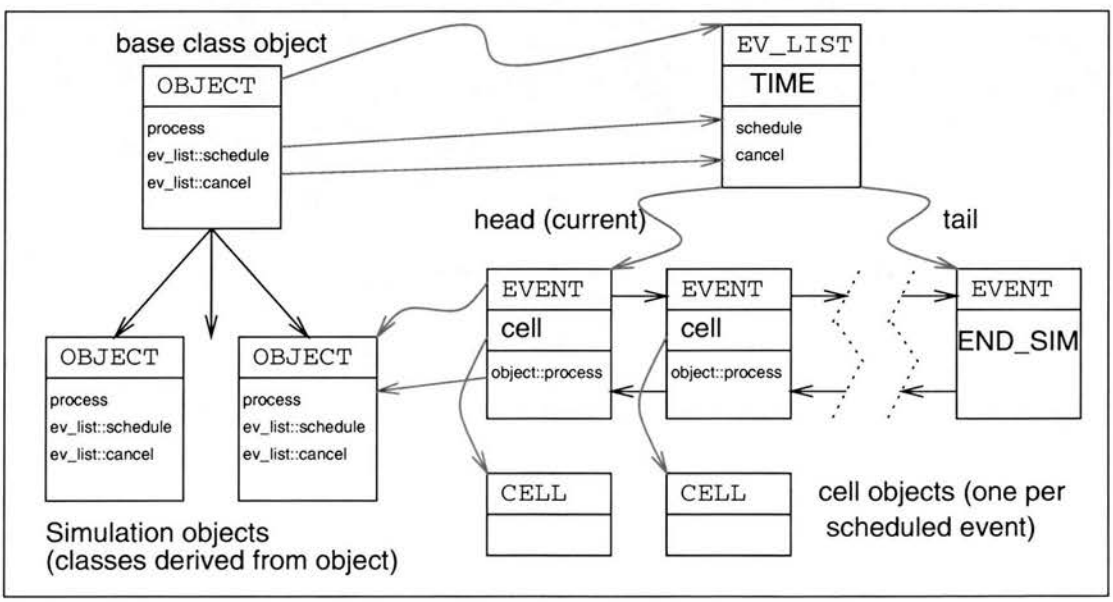


Figure 5.1: The object class relationships in the C++ cell-level simulator produced

As with the burst-level simulators, both the indexed doubly-linked event list and post-ordered tree event list implementations were integrated into the cell-level simulator. The choice of how one models simulation time can be made by altering a single C++ header file. Each part of the simulator refers to an abstract `time_type` variable type for representing simulation time. By suitably altering the C++ `typedef` statement which defines the actual type of `time_type`, either an integer or floating point type may be used to model the simulation time. In terms of modelling cells with events, each event models the *end* of a cell arrival at an object, thus marking the full availability of the cell for transmission to other objects. This choice was made to minimise the processing necessary by each simulation object.

The choice of physically buffering cells, or of directly scheduling output cells upon input cell arrival, can be made for each object individually rather than having to impose a general rule. This means that a direct comparison of the two approaches is possible in the context of an experiment running in the simulator. Although the cell-level simulator shares the simulation engine with the burst-level version, the simulation objects themselves are entirely different. As the behaviour of each object in the cell-level is very much simpler than at the burst-level, each object can be made to be as efficient as possible. The drawback is that each object has to process each and every cell passing through it, so maximum efficiency in the implementations will aid runtime performance. The cell and burst-level simulators also share the same random number generation code making directly comparable simulation runs straightforward.

5.3 Performance variables for the burst-level simulator

There are many factors which will influence the accuracy and performance of the integer-time burst-level simulator. As shown in Chapter 4, factors such as the sizes (in cells) of the bursts to be multiplexed, as well as the magnitudes of the ACTT values describing the bursts, have an impact on the accuracy of the techniques used. Other factors, such as the choice of the simulation event list management and of the integer arithmetic used in the core algorithms, will have an effect on the runtime performance achievable. Specifically, the following factors are of interest and will be the variables altered in the experiments performed in this Chapter.

Event list mechanism: With the choice available of an indexed event list or a

post-ordered tree for the global event list handling, either can be used to compare their runtime performances.

The variable type used for the global clock: A reason for using all integer arithmetic is to take advantage of the intrinsic performance advantage of integer arithmetic over the floating point equivalent.

Burst size in cells: One of the main reasons for choosing to abstract to the burst-level is that a number of cells are represented by just two messages, rather than by an individual message for each cell. Another advantage is that the cell level requires a decision as to whether to physically store cells in objects with buffer capability, or to schedule cells into the future to simulate the effect of buffering. The burst-level simulator should have comparatively better performance than the cell-level simulator when a burst describes a large number of cells, irrespective of whether storage or future scheduling is used at the cell level.

Replacing integer division: The other choice for the burst-level simulator is whether integer division is used when calculating burst ACTT values and when splitting bursts into fragments. The results in Chapter 4 show the theoretical reduction of costly integer division operations, in testbed versions of the core techniques, when division is replaced by the multiplication of reciprocals. However, when the techniques are used within real simulation objects running under the control of the event list, the effect may not be as profound.

ACTT multiplier: The results in Chapter 4 indicate that the accuracy of the integer burst-level algorithms improves if each ACTT is multiplied by some constant. However, the improved integer core techniques may suffer from reduced accuracy when dealing with large ACTT values. This is due to having to keep the value of the integer shift constant (used for calculating left shifted reciprocals) as small as possible to help prevent integer overflow in division calculations.

Event list bypass: If a simulation object receives its input from one other object (a "master"), and does not schedule any events for itself, it may be possible to make this object the *slave* of its master object. What this means is that the master object does not use the event list when scheduling events for its slave. Instead, the master holds a pointer to the `object::process` method of the slave, and directly passes the scheduled event to it. For

example, a queue without flow-control may be the slave of a multiplexer object, and this is shown in Section 5.7 for the burst-level simulator. The anticipated benefit of this approach is to reduce the total event count in the simulation and to cut out the overhead of scheduling events through the global event list. Both reductions should help improve the runtime performance of the burst-level simulator.

5.4 The simulators used

The following lists describe the simulators used in the experiments in this chapter. The shorthand names used to denote each simulator in the results are also shown. The cell and burst-level simulators used in this chapter were all developed in this work.

5.4.1 The cell-level simulators

The following simulators were used to produce the cell-level simulation results in this Chapter.

CL Simulator **CL** is a cell-level simulator which uses an indexed doubly-linked global event list. Global simulation time is represented by an unsigned 64-bit integer quantity (ie. `unsigned long long int` in C++).

CLt Simulator **CLt** is identical to simulator **CL** apart from the event list management which, in this case, is the post-ordered tree.

5.4.2 The burst-level simulators

The following simulators were used to produce the burst-level simulation results in this chapter.

BL Simulator **BL** is an integer-time burst-level simulator which uses an indexed doubly-linked global event list. Global simulation time is represented by an unsigned 64-bit integer variable. Standard integer division is used throughout the simulation objects implemented in this version of the simulator.

BLt Simulator **BLt** is identical to simulator **BL** in every detail apart from the event list management which utilises a post-ordered tree.

BL2 Simulator **BL2** is identical to simulator **BL** in every respect apart from the use of integer division by multiplication of reciprocals in the code for each simulator object.

BL2t Simulator **BL2t** is identical to simulator **BL2** apart from using a post-ordered tree for the global event list.

5.5 The experimental platform

All of the experiments conducted in this chapter were performed on the same workstation, which was configured as a shared compute server. The characteristics of the workstation are shown in Table 5.1. Although the workstation was a shared compute server, care was taken to perform experiments at times when no other users were running processes on the machine. This ensured consistent runtime performance measurements across the experiments.

Property	Description
Processor	400MHz Intel Pentium II
Memory	512Mb
Storage	8Gb IBM Ultra-SCSI 6Gb WD UDMA
Operating system	Linux (kernel 2.0.36)
C++ compiler	egcs-2.90.29
Compiler flags	-Wall -mcpu=i686 -march=i686 -O9 -malign-loops=2 -malign-functions=2 -malign-jumps=2 -mwide-multiply -fno-exceptions

Table 5.1: The configuration of the workstation used in the experiments in this Chapter

5.6 Multiplexing bursts

5.6.1 Introduction

The multiplexer has a very important job in both the cell and burst-level simulators. Streams of cells are merged onto one output link and thus the potential for introducing error to the cell stream statistics in the burst-level simulations is profound. The choice of simulation parameters, especially in the burst-level simulator, will have a great bearing on the accuracy of the output bursts from the multiplexer. The effect is less profound in the cell-level simulator, as the multiplexer can react to each and every input cell as it arrives to ensure that the output stream does not violate the maximum transmission rate permitted. There will be some difference to the results from a "real network" as the cell-level simulator is itself an abstraction of the behaviour of real hardware.

To achieve the same level of detail as at the cell level, the burst-level multiplexer must compute the ACTT for each output burst, and use burst splitting to fill the output `START` and `FINISH` messages with the correct number of cells for each input cell stream. Thus, this section compares the accuracy and performance of the burst-level multiplexer versus the cell-level multiplexer. The output of the cell-level multiplexer is taken to be the "ideal" which the burst-level technique must attempt to match as closely as possible. Although the cell-level simulator is also an abstraction of real behaviour, it is the most detailed form of simulation available. This makes the results of the cell-level simulations good candidates for comparison with the results of the burst-level simulations.

5.6.2 Cell-level multiplexing

The burst-level multiplexer is described in detail in Section 3.4 in Chapter 3. By contrast, the design of the cell-level multiplexer is very much simpler. Each arriving cell is represented by a `cell` message in an event whose time marks the *end* of the transmission of that cell. If the time difference between adjacent `cell` arrival messages is equivalent to the ACTT value in the burst-level simulator, the maximum output rate of the cell-level multiplexer can also be represented as an equivalent ACTT value. This permits exactly the same experiments to be performed in both the cell and burst-level simulators.

In order to produce cells at the multiplexer output which have adjacent transmission times not less than the minimum ACTT for the output link, the multiplexer needs to know the output ACTT and the time at which it last scheduled an output cell. There is a choice as to whether the multiplexer physically stores buffered cells, or directly schedules output cells at future simulation times to simulate buffering. If no storage is provided, the behaviour is as follows. When a new input cell arrives, if the difference between the current time and the last output transmission time (the "last send time") is greater than the output ACTT, the multiplexer can schedule an output `cell` event for that input cell. The last send time is then updated to become the current simulation time. If the time difference is less than the output ACTT, the last send time is incremented by one output ACTT, and the output `cell` event for the input cell is scheduled for the last send time. If storage is provided to model buffering in the multiplexer, queueing input cells are placed in the buffer. The sending of output cells is achieved by the multiplexer scheduling `WAKEUP` messages for itself, and sending the next cell in the buffer upon receipt of the message. Adjacent `WAKEUP` messages differ in time by the value of the minimum output ACTT for the multiplexer.

As with the burst-level multiplexer object, it would be possible to create the cell-level multiplexer as a compound object comprising of the multiplexer with a queue attached to the output port. This would offer no advantage over performing any appropriate cell buffering in the multiplexer object itself, as many cell transmission events would have to be processed to recognise the cells moving between the multiplexer and the queue object. The cell queueing techniques described in Section 5.7 can be easily integrated into the multiplexer, to police the output cell rate and perform any cell loss due to buffer overflow in the multiplexer buffer. The cell-level queue and multiplexer objects are fundamentally the same, as a multiplexer can be thought of as a queue which can accept cells from more than one source concurrently. The important actions are to ensure that the stream of output cells produced does not violate the minimum output ACTT set for the queue or multiplexer, and that any buffer overflow is modelled by discarding cells.

5.6.3 The Experiments

Figure 5.2 shows the simulation object structure of the multiplexing experiments performed. Both the cell-level and burst-level simulator experiments used the same object structure, and the same parameters for each simulation object type, such that a direct comparison of the results obtained could be made.

The job of each `RandSender` object, in both the cell and burst-level simulations, was to produce a stream of either bursts or cells respectively, each describing a single cell stream. The random number generator, and the parameters passed to each `RandSender`, were such that in equivalent experiments, the bursts produced in the burst-level simulator matched the equivalent *burst of cells* produced in the cell-level simulation (ie. the cells produced by the cell-level `RandSender` corresponded in time and ACTT with the `START` and `FINISH` messages for the same burst in the burst-level `RandSender`). The main choice for each `RandSender` object is that of how many bursts or cells are scheduled at any one time. In the case of the cell-level simulator, each `RandSender` was set to send up to 50 cells at any one time, from one burst, by scheduling the cells into the future. Cell production would then only restart upon the arrival of a `WAKEUP` message scheduled by itself for the next cell transmission time. The `RandSender` would also "sleep" between adjacent bursts through the use of a `WAKEUP` message scheduled for itself. The first 50 cells of the next burst would then be scheduled upon the arrival of this `WAKEUP` message. The idea is to keep the number of events *pending* (ie. events which have been scheduled but not yet processed) on the global event list low, to help keep the efficiency of event list operations high. The same logic is also appli-

cable in the case of the burst-level simulator. Here, each `RandSender` schedules the `START` and `FINISH` messages for one burst before "sleeping" by scheduling a `WAKEUP` message for the simulation time when the next burst `START` message need be generated.

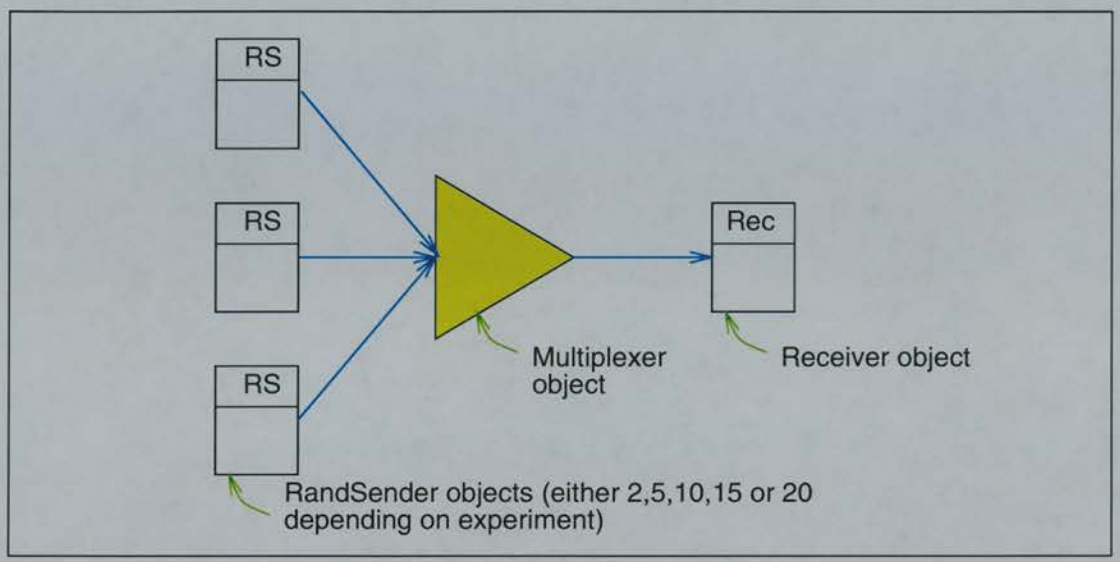


Figure 5.2: The simulator objects used in the multiplexing experiments

The job of the `Multiplexer` object in the burst-level simulation was to produce a single stream of multiplexed bursts at the output. Each input burst contained a single cell stream, thus the maximum number of component cell streams in each output burst equalled the total number of `RandSender` objects defined in each simulation run. In the cell-level simulation runs, the `Multiplexer` object produced a single stream of cells at the output from the cells arriving at its input. As the purpose of these experiments was to examine the performance of multiplexing only, no cell buffering was required for either the cell or burst-level multiplexers. This was achieved by setting the minimum output ACTT for each object, such that the output bandwidth was larger than the combined input bandwidth if each `RandSender` was transmitting concurrently. In the case of the cell-level multiplexer, some buffering *is* required, as more than one input cell may arrive at the same time. Rather than provide internal storage, concurrent cells were handled by scheduling cells into the future such that the minimum output ACTT was enforced.

The `Receiver` object had the job of collating the burst statistics for each simulated cell stream. This was achieved by giving each `RandSender` a unique cell stream identifier and by giving each burst a sequence number. The combination of the identifier and the sequence number gave each burst produced a unique

identification number. This information, whether carried by a cell or by a `START` or `FINISH` message, could be used to produce timing statistics for each burst of cells for each cell stream received at the `Receiver`. In the case of the burst-level simulations, the receiver produced statistics for each cell stream based on the fragments described by each arriving pair of burst `START` and `FINISH` messages. Each component stream in a multiplexed burst is marked with the burst unique identifier which enables the statistics to be gathered. At the cell level, each `cell` is marked with the unique burst identifier for the burst of which it is part. This enables the `Receiver` in the cell-level simulator to collate timing statistics for each cell stream transmitted in the simulation.

It would be impossible to explore the entire parameter space of all the simulator options possible for multiplexing, as the accuracy and performance of either the cell or burst-level techniques is heavily dependent on the experiment being performed. Instead, the experiments performed varied the number of cell streams, the number of cells per stream, the ACTT of each burst and the interarrival time distribution between bursts. The same experiments were performed for each of the simulator types introduced in Section 5.4. This allowed for reasonable comparison of the various performance related factors in each simulator type. Random number distributions were used to provide the burst ACTT, the number of cells in each burst and the delay between adjacent bursts in each cell stream.

The steps taken to produce a burst in each `RandSender` object (for each simulation run) were as follows:

1. Produce a time delay before the start of the next burst which is used as the burst interarrival time. A simple means of doing this was to produce a "dummy" burst (ie. draw a value from the cells' distribution and multiply it by a value drawn from the ACTT distribution) whose duration was made to be the burst interarrival time. In order to add some variability to the burst interarrival times, the "dummy" burst duration was multiplied by a number drawn from a floating point distribution. The result (rounded to an integer) was then used as the delay before the next actual burst of cells was produced.

2. Produce a burst of cells. Once again, the ACTT for the burst was drawn from the ACTT distribution, while the number of cells for the burst was drawn from the cells' distribution. Depending on the simulation type:

    - In the cell-level simulation, up to 50 cells (less if the total number of cells for the burst is less than 50) are individually scheduled for the

cell multiplexer. A WAKEUP message is scheduled either for the next cell transmission time or for producing the next inter-burst delay.

- In the burst-level simulation, the START and FINISH messages for the burst are appropriately scheduled, and a WAKEUP message is scheduled for producing the next inter-burst delay.

3. Repeat until the required number of bursts have been produced.

Table 5.2 shows the experimental parameters used for each simulation run. The *Randint*(A,B) distribution produces a random integer X in the range $A \leq X \leq B$, where each value X has equal probability of selection. The *Uniform*(C,D) distribution produces a random floating point number Y in the range $C \leq Y \leq D$, where each value Y has equal probability of selection. The "Gap mult" distribution shown provided the real number with which to multiply the duration of the "dummy" burst to produce a delay between adjacent bursts. By altering the upper limit of the "Gap mult" distribution, two sets of results were produced for "sparse" and "dense" burst production. "Dense" burst production ensured that the burst interarrival delay was some fraction less than 1 of the dummy burst, whereas "sparse" production allowed for the delay to be up to 10 times larger than the dummy burst duration.

Setting	"Dense" burst arrivals	
Streams	2,5,10,15,20	2,5,10,15,20
Cells	*Randint*(10,1000)	*Randint*(100,10000)
ACTT	*Randint*(100,10000)	*Randint*(100,10000)
Gap mult	*Uniform*(0.0,1.0)	*Uniform*(0.0,1.0)
No. bursts	10000	10000
Setting	"Sparse" burst arrivals	
Streams	2,5,10,15,20	2,5,10,15,20
Cells	*Randint*(10,1000)	*Randint*(100,10000)
ACTT	*Randint*(100,10000)	*Randint*(100,10000)
Gap mult	*Uniform*(0.0,10.0)	*Uniform*(0.0,10.0)
No. bursts	10000	10000

Table 5.2: The experimental parameters

5.6.4 Results

For the multiplexing experiments, an analysis of the comparative runtimes of the various techniques is presented first, followed by an analysis of the comparative accuracy between the techniques. The runtime experiments were undertaken in an

independent batch, with minimal processing of burst data at the `Receiver` object. A second batch of experiments, instrumented to provide information necessary for accuracy analysis, was then performed for the same simulation parameters. The accuracy results start from Section 5.6.4.2.

5.6.4.1 Runtime comparisons

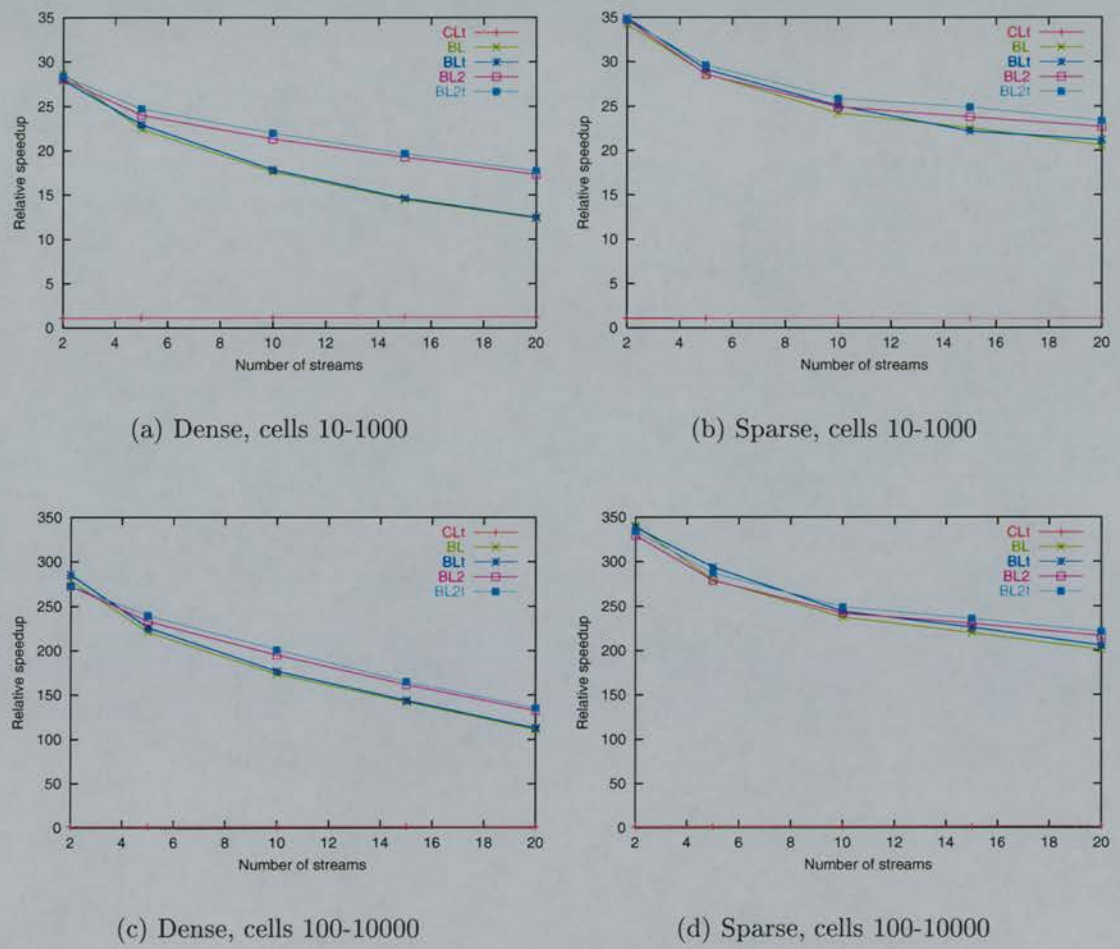
Experiments were performed using the parameters in Table 5.2 for each of the simulators listed in Section 5.4. All of the experiments were performed on the same workstation, with effort made to ensure that it was unloaded by other user processes, so that no bias was introduced into the runtime measurements. The raw results of each experimental run are shown in Table 5.3.

"Dense" burst production and burst cell size 10-1000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	49	45	1.7	1.7	1.7	1.7
5	132	118	5.9	5.8	5.5	5.3
10	279	244	15.9	15.7	13.1	12.7
15	441	373	30.5	30.2	22.9	22.4
20	617	519	49.9	49.6	35.7	34.9
"Dense" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	496	455	1.8	1.7	1.8	1.8
5	1308	1176	5.9	5.8	5.6	5.5
10	2766	2421	15.9	15.7	14.2	13.8
15	4334	3707	30.6	30.2	26.9	26.3
20	6056	5185	54.7	53.9	45.8	44.9
"Sparse" burst production and burst cell size 10-1000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	49.9	46.2	1.4	1.4	1.4	1.4
5	127.3	119.9	4.4	4.3	4.4	4.3
10	260	247	10.7	10.4	10.4	10.1
15	4.6	386	18.1	13.3	17.1	16.3
20	549	523	26.7	25.9	24.25	23.5
"Sparse" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	493	455	1.4	1.4	1.5	1.5
5	1259	1168	4.5	4.3	4.5	4.4
10	2560	2381	10.8	10.5	10.6	10.3
15	3989	3700	18.2	17.7	17.4	16.9
20	5373	4976	26.8	26.2	24.8	24.3

Table 5.3: The actual runtimes (in seconds) of each experiment performed in the multiplexer simulation tests

Figure 5.3 presents four figures showing the relative speedup of each simulation

program, when compared to the results of simulator **CL** used as a baseline (ie. the runtime for each experiment in the **CL** simulator is equivalent to a speedup of 1.0). In each experiment performed, simulator **CLt** (cell level using a post-ordered global event tree) was faster than the indexed doubly-linked global event list version (**CL**), hence the inclusion of a speedup plot for simulator **CLt** in each graph. This implies that some fine-tuning of the maximum number of events per event index in simulator **CL** may be possible, although the relative speedup was always just sightly larger than one.



(a) Dense, cells 10-1000

(b) Sparse, cells 10-1000

(c) Dense, cells 100-10000

(d) Sparse, cells 100-10000

Figure 5.3: The relative runtime speedups for each simulator in the multiplexer experiments

As can be seen from each group of experimental results presented in Figure 5.3, the burst-level simulators offered a relative speedup over their cell-level counterparts. Of particular relevance to the magnitude of the speedup was the distribution chosen for the number of cells in each burst. Table 5.4 shows the total number of events processed in each simulation run for "dense" burst production,

along with the maximum number of events pending on the global event list at any one time. Table 5.5 presents the same data for the "sparse" burst production experiments. As would be expected, due to choosing the same random number generator and the same parameters for each experiment, the event counts are equal according to the simulator type used for each experiment type. As can be seen from the event counts for the cell-level simulations, the number of cells simulated has a great effect on the total number of events processed, as would be expected. In the case of the burst-level simulators, the total number of events to process is not as clearly related to the number of cells simulated, as a burst is represented solely by a pair of `START` and `FINISH` messages. Other factors, such as the degree of *burst fragmentation* (ie. the total number of burst fragments an original burst is split into by the actions of being multiplexed and queued) and the total number of cell streams simulated have a bearing too. The level of burst fragmentation in each simulation run is discussed further in Section 5.6.4.5.

Streams	Cells 10-1000		Cells 100-10000	
	CL & CLt	BL & BL2	CL & CLt	BL & BL2
2	20429788 (108)	126420 (11)	204209729 (108)	126416 (11)
5	51218598 (264)	348651 (22)	511966269 (264)	348635 (20)
10	102147443 (523)	699838 (37)	1021036910 (523)	699840 (34)
15	153155531 (783)	1049855 (50)	1530901611 (783)	1049857 (47)
20	204047269 (1042)	1399860 (62)	2039601485 (1042)	1399862 (62)

Table 5.4: The total event counts and maximum number of events pending (in brackets) for each simulator when performing "dense" burst multiplexing

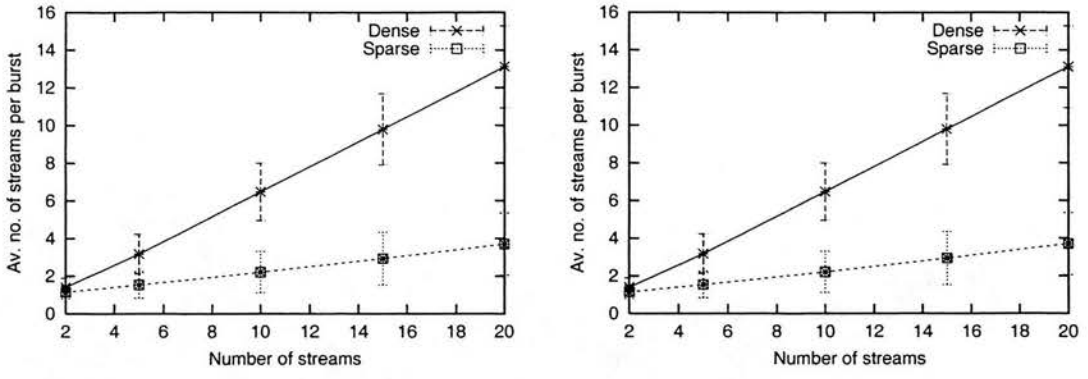
The burst-level simulator which gave the best performance in nearly every experiment was **BL2t**, which utilised integer division by reciprocal multiplication and a post-ordered tree event list. However, the performance advantage of **BL2t** over the event indexed **BL2** was minimal, suggesting that the small maximum number of events pending (see Tables 5.4 and 5.5) suited the indexed doubly-linked global event list. The burst-level simulators using integer division by multiplication of reciprocals performed better than the ones using standard integer division, with an especially noticeable difference in the "dense" burst production experiments. Here, the burst-level simulators using the integer division improvements from Chapter 4, offered up to 50% greater speedup than provided by the burst-level simulators using standard integer division (for 20 streams in

Streams	Cells 10-1000		Cells 100-10000	
	CL & CLt	BL & BL2	CL & CLt	BL & BL2
2	20429788 (108)	106658 (11)	204209729 (108)	106608 (9)
5	51218598 (264)	301339 (20)	511966269 (264)	301271 (20)
10	102147443 (523)	660030 (37)	1021036910 (523)	659956 (35)
15	153155531 (783)	1025223 (52)	1530901611 (783)	1025191 (50)
20	204047269 (1042)	1386340 (69)	2039601485 (1042)	1386548 (65)

Table 5.5: The total event counts and maximum number of events pending (in brackets) for each simulator when performing "sparse" burst multiplexing

the "dense" experiments with the burst cells size ranging from 10 to 1000). In the "dense" set of experiments, bursts arrive in rapid succession and thus many streams will need to be merged at the output for each output burst produced. Production of a multiplexed output burst requires calculation of the burst ACTT along with burst splitting to provide the cells counts. Both of these operations were shown to have potential performance improvement through the use of integer division by multiplication of reciprocals. The potential improvement, over the standard integer division technique, increases with increasing number of input cell streams. This is shown in the increasing difference in speedup between **BL** and **BL2** with increasing number of cell streams. In the "sparse" burst production experiments, the number of output streams to merge at any one time will be lower, and thus the potential performance benefit of the improved integer techniques is not as pronounced. Figure 5.4 shows the average number of cell streams per multiplexed burst received by the `Receiver` object for the "dense" and "sparse" burst production experiments for both ranges of burst cell size. The difference in the work required of the multiplexer for the "dense" and "sparse" experiments is clear. The lower number of cell streams to multiplex in the "sparse" experiments also explains the relative difference in speedup to the equivalent "dense" experiments using the same burst size distribution.

The actual overhead of performing burst multiplexing will be complicated with execution stalls in the machine code, caused by loop branching and conditional statement processing. These stalls may allow the compiler, or even the dynamic scheduler in the processor, to hide the cost of the integer divide instructions, thus limiting the performance difference between **BL2** and **BL**. However, removing the integer divides from the burst-level multiplexer results in the **BL2** and **BL2t**



(a) No. cells per burst 10-1000

(b) No. cells per burst 100-10000

Figure 5.4: The average number of cell streams per burst received (with standard deviation shown) at the `Receiver` object for both ranges of burst size

simulators giving shorter runtimes than the integer division **BL** and **BLt** simulators, especially for the larger number of cell stream experiments. This difference is pronounced in the "dense" burst production experiments, as the average number of streams to merge for each multiplexer output burst is larger, and this requires more processing. As this processing can use the integer improvements, a performance improvement is seen relative to the implementation which uses standard integer arithmetic.

As can be seen from the four sets of results, the speedup offered by the burst-level simulators, over the cell-level simulators, reduces with increasing stream count. This is understandable, as the greater the number of streams to merge, the more work the multiplexer has to do for each output burst message pair. When the number of cells per input burst is small, and the interarrival time between adjacent input bursts in the same stream is small, the multiplexer will have to produce many output bursts, each describing small fragments of the original input bursts. At some point, it may be possible that each output burst contains so few cells that it would be no longer advantageous to use the burst level over the cell level. In this scenario, simulating each cell individually may be more efficient. Of course, the "decomposition" of the burst-level to the cell-level may be short-lived in an experiment. The runtime gains possible when a burst describes many cells may more than make up for any transient poor performance encountered at some point in a simulation run.

5.6.4.2 Performing the accuracy analysis

For each of the experiments whose runtime results were presented in the previous section, data regarding each burst arriving at the `Receiver` object was recorded in another batch of experiments using identical parameters. In this batch of experiments, the `Receiver` object was designed to keep statistics for each cell stream arriving from the multiplexer on a "per burst" basis. In the burst-level simulations, this involved using the burst unique identifier in each component stream in each burst to determine the source cell stream. In the cell-level simulations, the unique identifier in every cell was used to determine the cell stream and burst number for an arriving cell.

In the case of the burst-level simulations, the data recorded was the start time, finish time, total number of cells and total number of burst *fragments* for each burst in each cell stream. The number of burst fragments for a particular burst is equal to the number of bursts to arrive at the `Receiver` object which carry a component stream containing cells from that burst. Hence, the start time for the burst is the time of the first `START` message to arrive carrying a fragment of the burst. The finish time for the burst is the time of the last `FINISH` message to arrive carrying a fragment of that burst. If the time of the first `START` message for the burst was $T_{\texttt{STA}}$, the time of the last `FINISH` message for the burst was $T_{\texttt{FIN}}$ and the number of cells to arrive was N, then the effective ACTT value for the burst, $ACTT_B$, can be computed with Equation 5.1.

$$ACTT_B = \frac{T_{\texttt{FIN}} - T_{\texttt{STA}}}{N} \tag{5.1}$$

For the cell-level simulations, the data recorded was the arrival time of the first cell from each burst, the unique identifier for each burst, the arrival time of the last cell for each burst and the total number of cells received for the burst. From this data, an effective ACTT for the burst of cells can be calculated. If N cells arrive for a particular burst, with the first cell arriving at T_S and the last cell arriving at T_F, then the effective ACTT ($ACTT_B$) for the burst of cells can be calculated with Equation 5.2.

$$ACTT_B = \frac{T_F - T_S}{N - 1} \tag{5.2}$$

As each cell arrival event marks the *end* of transmission of a cell, the ACTT calculated in Equation 5.2 requires the divisor to be equal to the total number of cells to arrive minus 1. This is the average transmission time for the remainder of the cells in the burst. To include the first cell, the computed ACTT can be

used to adjust the burst start time, such that the transmission time for the first cell is included. As T_S is the time of the end of the first cell transmission for the burst, an estimate of the time of the start of the burst, T'_S, is calculated with Equation 5.3.

$$T'_S = T_S - ACTT_B \tag{5.3}$$

The estimated burst start time, the burst finish time and the estimated burst ACTT value for each burst in the cell-level simulator could then be compared with the corresponding values for each burst with the same unique identifier in the equivalent burst-level experiments. One point to note is that the ACTT multiplier used in each of the burst-level experiments was 1000, thus, a suitable scaling of absolute time recordings is required before comparison with the cell-level results.

In the following sections, comparisons are made between the results from the **BL** and **BL2** simulators and the results of the **CL** simulator, regarded as having the "ideal" result set. Results were also recorded for the **CLt**, **BLt** and **BL2t** simulators, but each result set was identical to the results from the indexed event list experiment in each case. This is as expected, as the only difference was the event list mechanism which should have no bearing on the accuracy of the results.

5.6.4.3 Burst start time comparison

The first analysis performed was to compare burst start times between the burst-level and cell-level experiments with the same simulator parameters. For each cell stream in each experiment, the mean absolute time difference between the burst-level and cell-level burst start times was computed. For each experiment set (eg. one experiment set was "dense" burst production with the burst cell size taken from the range 10 to 1000, calculated for 2, 5, 10, 15 and 20 individual cell streams), the lowest and highest mean differences were recorded. The raw data (also showing the standard deviation for each mean) can be found in Tables 5.6 and 5.7. The range of mean burst start delays incurred in the burst-level simulations (relative to the cell-level version), for both burst size distributions simulated with "dense" burst production, is shown in Figure 5.5.

The immediately obvious feature from Figure 5.5 is the large difference between the mean burst start delays in the **BL** and **BL2** simulators. This is initially surprising, as the analysis in Chapter 4 shows that the introduction of the revised integer techniques makes very little difference to the accuracy of the multiplexer ACTT and burst fragment calculations. However, the accuracy of the revised in-

"Dense" burst production and burst cell size range 10-1000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	5084.3 (3311.4)	5196.1 (3379.8)	5091.9 (3314.1)	5197.2 (3376.7)
5	9334.4 (7151.7)	9470.2 (7226.2)	9734.0 (7515.3)	9849.3 (7568.3)
10	16238.9 (10183.3)	16476.7 (10376.9)	43490.1 (27927.1)	44130.8 (28070.6)
15	12882.3 (8352.9)	13200.1 (8445.1)	118761.2 (55210.7)	120946.3 (54982.0)
20	8434.3 (5379.5)	8577.3 (5461.1)	81823.4 (44854.2)	83950.7 (45208.4)
"Dense" burst production and burst cell size range 100-10000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	5007.2 (3255.1)	5107.4 (3291.3)	5039.5 (3319.7)	5126.8 (3339.9)
5	6406.7 (3890.7)	6494.7 (3966.8)	9221.2 (6766.2)	9375.5 (6949.4)
10	3929.1 (2055.5)	4017.0 (2180.8)	103530.5 (79985.6)	106087.6 (79680.2)
15	3066.8 (1626.4)	3154.9 (1784.9)	178831.8 (103924.4)	184320.2 (105451.1)
20	2753.2 (1501.9)	2828.1 (1623.6)	254320.6 (152535.7)	262151.4 (153554.3)

Table 5.6: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst `START` time (cf. the cell-level results) for the burst-level simulators

teger techniques is dependent on the values used in the calculations. Left shifted reciprocal ACTT values are used to calculate the output burst ACTT value in the multiplexer in simulator **BL2**, and the accuracy of the left shifted reciprocal method *decreases* with increasing ACTT value. In the experiments performed, the ACTT values for each burst could range from 100 to 10000, but each value was also multiplied by 1000 to aid accuracy. This resulted in an absolute range of 100000 to 10000000 for each ACTT value in the experiments. Coupled with an integer constant of 2^{38} as the *shift constant* used for calculating shifted reciprocal values, this limits the accuracy of the revised integer techniques. It is necessary to keep the integer shift constant as low as possible to aid prevention of integer arithmetic overflows when performing integer division by reciprocal multiplication. To assess the impact of the reduced burst ACTT accuracy in simulator **BL2**, the experiments were conducted again for the "dense" burst production

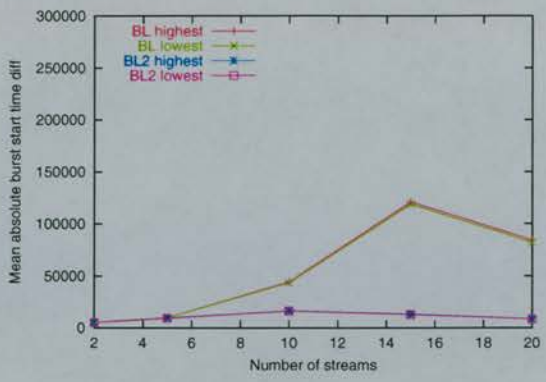
"Sparse" burst production and burst cell size range 10-1000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	3060.8 (2193.5)	3110.0 (2202.7)	3059.2 (2188.9)	3110.6 (2203.7)
5	4100.3 (2743.7)	4215.9 (2791.7)	4101.3 (2743.5)	4214.4 (2791.8)
10	5023.7 (3076.0)	5121.4 (3194.4)	5037.6 (3092.2)	5133.5 (3223.4)
15	5675.9 (3667.9)	5819.3 (3836.6)	5728.1 (3718.2)	5869.5 (3886.2)
20	6324.8 (4322.4)	6463.5 (4405.5)	6475.3 (4455.4)	6622.9 (4564.4)
"Sparse" burst production and burst cell size range 100-10000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	3036.8 (2133.9)	3072.9 (2182.6)	3041.0 (2142.2)	3072.4 (2183.5)
5	4093.9 (2718.9)	4139.1 (2760.1)	4108.8 (2744.9)	4162.3 (2794.3)
10	4855.7 (2949.1)	4937.7 (2954.9)	5014.9 (3114.3)	5088.7 (3104.3)
15	5162.6 (3179.3)	5285.6 (3321.5)	5663.5 (3632.2)	5770.5 (3753.8)
20	5297.8 (3326.3)	5416.6 (3448.9)	6393.1 (4389.4)	6543.4 (4555.7)

Table 5.7: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst START time (cf. the cell-level results) for the burst-level simulators

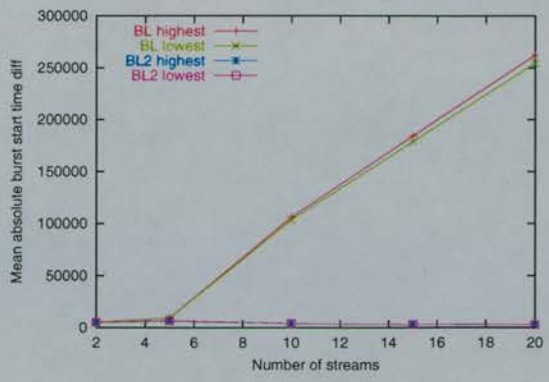
case where the burst size range was 100 to 10000 cells. The simulator had extra code added to compare the ACTT calculated for each output burst with an "ideal" ACTT for the burst calculated with Equation 5.4, where N streams are being multiplexed and $ACTT_i$ is the ACTT value for stream i. The result computed by Equation 5.4 is the integer ceiling of the result, as the burst ACTT is designed to be an overestimate (as described in Chapter 3), so this gives the best comparison with the integer value computed.

$$ACTT_{ideal} = \left\lceil \frac{1.0}{\sum_i^N \frac{1.0}{ACTT_i}} \right\rceil \tag{5.4}$$

For each experiment (ie. 2, 5, 10, 15 or 20 cell streams), the ACTT calculated by the integer technique was compared with the ideal ACTT calculated for the same output burst, and a record was kept of any discrepancy. Table 5.8 presents the percentage of output bursts for which the integer ACTT disagreed with the



(a) No. cells per burst 10-1000



(b) No. cells per burst 100-10000

Figure 5.5: Lowest and highest mean absolute burst start time differences (in units of simulation clock time divided by the ACTT multiplier) for the burst-level simulations in the "dense" burst production experiments

ideal ACTT for each experiment. Also presented is a mean ACTT ratio calculated as the integer ACTT divided by the ideal ACTT when the two differed. The same analysis was performed for the same experiments in simulator **BL**. Here, there was virtually no difference in every experiment between the integer burst ACTT computed and the ideal ACTT, hence the results are not shown.

Streams	% differences	Mean ACTT ratio	sd
2	38.229	0.9999906	0.0000058
5	92.718	0.9999899	0.0000050
10	98.497	0.9999902	0.0000037
15	98.555	0.9999903	0.0000031
20	98.498	0.9999904	0.0000028

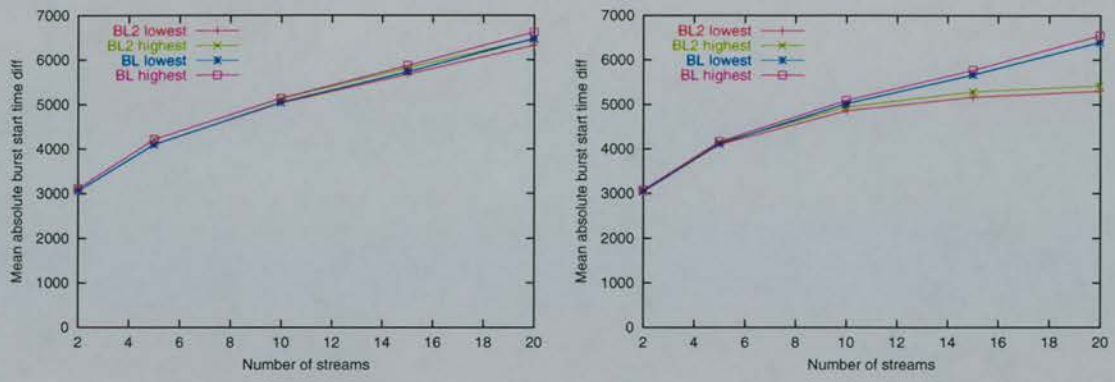
Table 5.8: The percentage of ACTT underestimates in **BL2** for "dense" burst production and burst size range 100-10000 cells. The mean ACTT ratio (with standard deviation) for each underestimate is also shown.

As can be seen from Table 5.8, the magnitude of the disagreement was small, as the ACTT ratio was near to 1.0 with a small standard deviation. However, this *underestimate* (as the mean ACTT ratio was less than 1.0) is enough to help reduce the average burst start time difference. In the "dense" burst production experiments, bursts arrive relatively frequently at the multiplexer, and so many output bursts need to be created. The work performed for each output burst increases with increasing the number of input streams which have to be merged. Burst fragments need to be computed to provide the cell counts for each burst, and these counts are coupled with the calculated ACTT for each burst to calcu-

late the duration of the output burst. If the burst ACTT is an *overestimate*, even by a relatively small amount, the net effect is to introduce some delay into the scheduled times of output `START` and `FINISH` messages (as the messages must be generated in strict order of increasing simulation time). This is because the calculated output burst message schedule times will not necessarily coincide with the arrival times of input `START` and `FINISH` messages which trigger the production of the output burst messages.

If a burst-level multiplexer is "busy", ie. coping with frequent burst arrivals from many inputs, there is no time to recover any delay incurred at the output due to output ACTT overestimation. This has the effect of compounding message delay times at the output until a natural break occurs in the input burst streams. At this point, the next scheduled output `START` message can be sent at the time actually calculated by the multiplexer, as the delay can be recovered when the output is not transmitting bursts. The effect is most dramatic in the "dense" experiments where the burst size ranges from 100 to 10000 cells. The **BL** multiplexer cannot easily recover from delays introduced at its output, and thus the average start time difference increases with increasing number of input streams. The multiplexer in **BL2**, on the other hand, consistently underestimates the burst ACTT by a small fraction, and this goes some way to offsetting any delay at the multiplexer output. This can be seen in Figure 5.5. As long as the multiplexer uses the values it calculates for burst cell counts and the burst ACTT, either the under or overestimate of the burst ACTT is tolerated, just as long as the burst generated *at least* fills the time period required for the output burst. Each multiplexer object ensures that the basic rules for bursts, as described in Section 3.3.2, are maintained for the ACTT value computed. Other simulation objects rely on the details contained in the burst `START` message, and these details will be correct with respect to the subsequent `FINISH` message generated by the multiplexer.

Considering the large difference in the average start time differences for the burst-level simulators in the "dense" experiments, the picture is very different in the "sparse" experiments. The range of mean burst start delays for each set of "sparse" burst production experiments can be seen in Figure 5.6. Here, the difference between the two types of burst-level simulator is very much reduced, along with the actual magnitude of the time difference. This confirms that the multiplexer objects in **BL** and **BL2** are able to recover from any burst message scheduling delays at the output due to the sparse nature of the input burst arrivals.



(a) No. cells per burst 10-1000

(b) No. cells per burst 100-10000

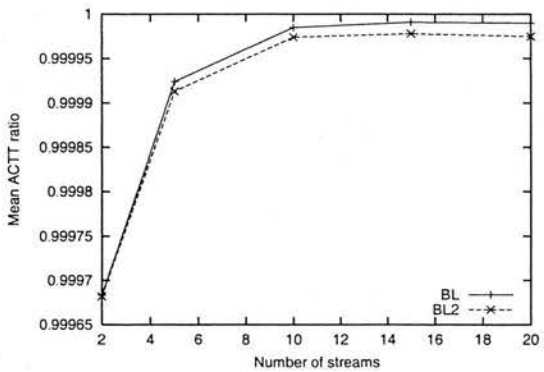
Figure 5.6: Lowest and highest mean absolute burst start time differences for the burst-level experiments in the "sparse" burst production experiments

The results presented above are the range of average burst start time differences for each burst-level simulator when compared to the cell-level simulator results. Although the average gives an insight into the behaviour, it must be taken in context with the large standard deviations (relative to the magnitude) of each result (see Tables 5.6 and 5.7). This ties in with the multiplexer both suffering from, and recovering from, a cumulative delay in burst message scheduling at its output port. When the delay is small, this implies that the multiplexer has little delay at its output port, whereas a large delay shows that the multiplexer has been "busy", and thus has a large cumulative scheduling delay due to this. It should also be noted that the burst start time in the cell-level results used is an estimate based on the statistics of the burst. It is possible that using the average ACTT calculated for each cell in the burst bar the first to arrive, and then using this to set the start time, may be an over-simplification. However, any inaccuracy introduced will be limited, especially in the experiments where many cells are scheduled in each burst, thus providing a larger sample with which to calculate the effective ACTT for the burst of cells.

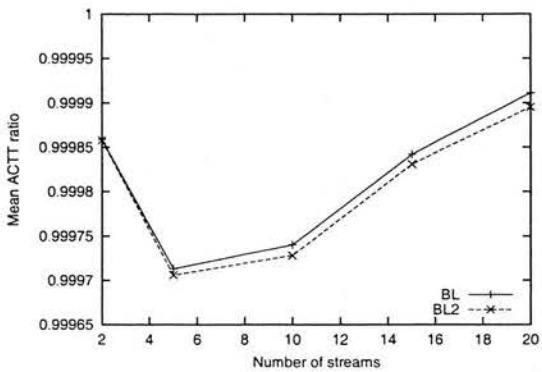
5.6.4.4 Burst ACTT comparison

The next part of the analysis of the results was to look at the comparison between burst ACTT values in the burst and cell-level experiments. This was achieved by computing the effective ACTT for every burst in the experiments performed in both the cell and burst-level simulators, and then by computing an ACTT ratio for comparison purposes. For every burst received by the `Receiver` object in the burst-level experiments, the ACTT ratio for that burst was defined to be

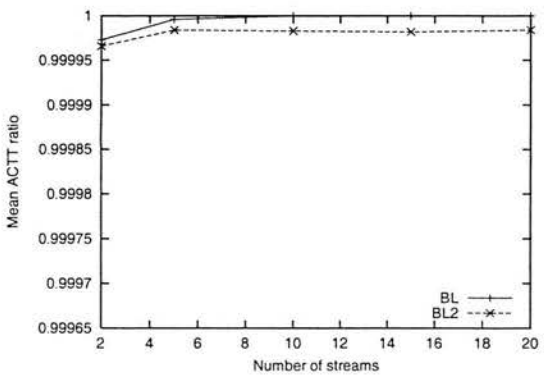
the effective ACTT divided by the effective ACTT for the burst with the same unique identifier in the cell-level experiment with the same parameters. In order to judge the effect of the multiplexing process over all of the bursts simulated, the ACTT ratios for every burst produced in a simulation were considered in each analysis. The other option would be to consider the effect on each individual cell stream, but this would only give results for a particular stream. Analysis for every burst produced gives an indication of the general level of accuracy obtainable by the technique. The mean (with standard deviation) ACTT ratios, comparing simulator results from **BL** and **BL2** with the results from simulator **CL** can be seen in Table A.1 in Appendix A. Figure 5.7 presents plots of the mean ACTT ratio results for both burst size distributions in both the "dense" and "sparse" burst production experiments.



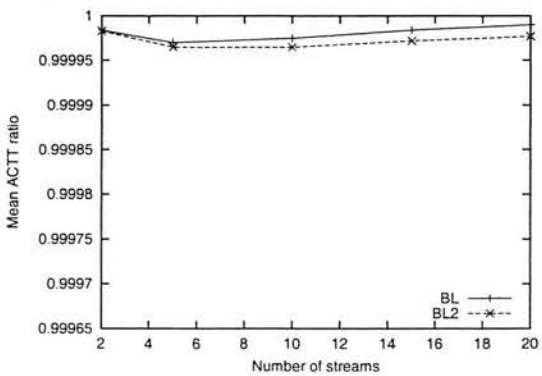
(a) Dense, cells 10-1000



(b) Sparse, cells 10-1000



(c) Dense, cells 100-10000



(d) Sparse, cells 100-10000

Figure 5.7: Mean ACTT ratios for all of the bursts received at the `Receiver` object in the multiplexer experiments

As can be seen from Figure 5.7, there is little difference between the mean

ACTT ratios for each type of burst-level simulator. However, the mean ACTT ratio is consistently *lower* in the **BL2** experiments, which supports the fact that the revised integer techniques produce an underestimate of the burst ACTT value at the multiplexer. As the ACTT value for each burst in each cell stream is taken as the duration of all the fragments of the burst divided by the total number of cells, "shorter" fragments (due to a burst ACTT underestimate) will reduce the final effective ACTT value for the burst. The mean ACTT ratios in the experiments with burst sizes ranging from 100 to 10000 cells show a consistently high ACTT ratio. This is in part due to the length of each burst, as any error introduced by the multiplexing process is averaged over the larger number of cells (cf. the burst size 10 to 1000 experiments). For the lower burst size distribution experiments, there is an improvement in the mean ACTT ratio with an increase in the number of component streams. This is especially noticeable in the "dense" burst production experiments, although the scale of the absolute difference is very small. The underestimate is most likely due to various fragments of each burst containing cells allocated by cell contribution fractions. In this case, the duration of the multiplexed burst carrying a fragment of a burst will have a shorter duration than the original ACTT of the burst multiplied by the number of cells carried from that burst. This may reduce the total duration of the burst and hence the effective ACTT. With more streams to merge, the influence of cell fractions will be divided more fairly between the component cell streams, especially when the multiplexer is required to deal with dense burst arrivals. In this case, each stream will have fragments containing a cell allocated by cell contribution fractions, but will also incur delay due to multiplexed bursts containing fractional cell contributions of other streams. The net effect is to reduce the underestimate for all streams. Once again, the greater the number of cells in each burst, the lesser the effect.

5.6.4.5 Burst fragmentation

A factor which will have an effect on the runtime performance of the burst-level simulators, as well as on the burst start time delay due to cumulative delays at the multiplexer output, is the degree of fragmentation each burst experiences. To get a feel for the average number of burst fragments each burst is split into in the experiments, the total number of fragments received for each burst at the `Receiver` was recorded. An average could then be computed. Due to the nature of the random number generator used in the experiments, the average fragmentation was found to be very close irrespective of the burst cell size distribution chosen. The results for the "dense" and "sparse" burst production experiment sets were

different, as would be expected. The similarity is because the same random seed was passed to the random number generator for each experiment, such that the result on drawing a number from the *Randint*(10,1000) distribution at a certain iteration, differed to the result from the *Randint*(100,10000) by approximately a factor of 10 for the *same* iteration. This meant that although the actual number of cells varied, the burst cell sizes, and the interarrival delay between them differed only by a constant factor. This means that the fragmentation of bursts was almost identical between the different burst size distributions in both the "dense" and "sparse" experiment types. The average (with standard deviation) burst fragmentation for the "dense" and "burst" experiment batches (results taken from the **BL** experiments) can be seen in Figure 5.8.

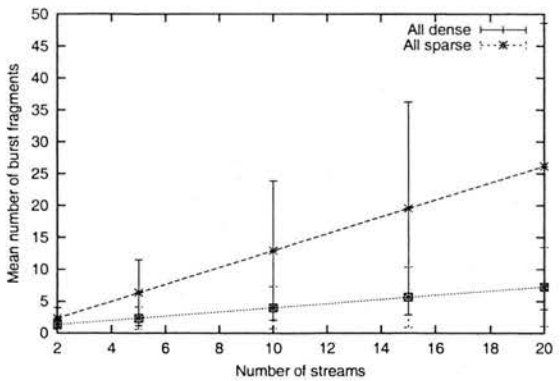


Figure 5.8: The mean final burst fragmentation for each multiplexer experiment type

Even with the relatively large standard deviations of the mean fragmentation results, Figure 5.8 shows that the average burst fragmentation in the "sparse" experiments was less than in the "dense" experiments. As would be expected, the degree of fragmentation increases with an increasing number of cell streams, with the fragmentation increasing more sharply in the "dense" burst production experiments. When the fragmentation is smaller, the multiplexer is doing less work and this helps improve the runtime performance of the burst-level simulator. This can be seen in the relative speedup differences for the burst-level simulators between the "sparse" and "dense" experiment sets in Figure 5.3 on page 141. The lower burst fragmentation is also a contributory factor to the low average burst start delay times seen in Figure 5.6 on page 151.

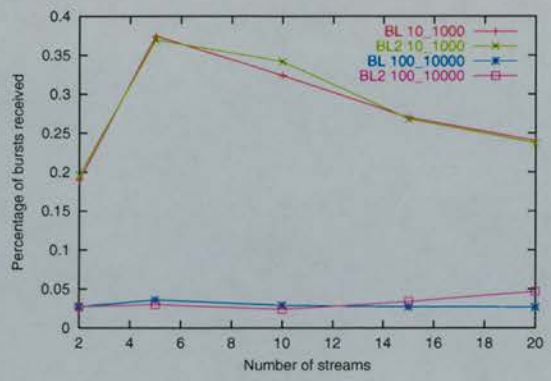
5.6.4.6 Zero cell length and "overlong" bursts

Other effects on bursts due to multiplexing are the production of *zero cell length* bursts and *overlong* bursts. A zero cell length burst is a pair of `START` and

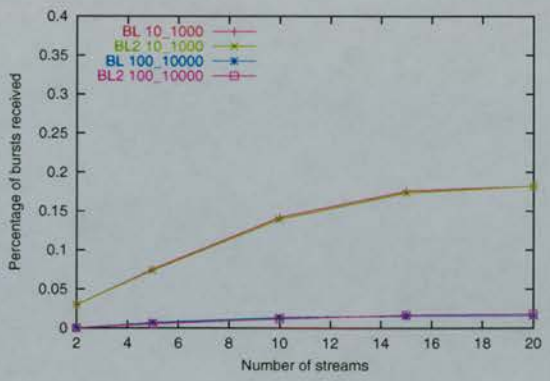
FINISH messages in which the START message describes a burst with a maximum cell payload of greater than zero cells, but the subsequent FINISH message details that the burst contains zero cells in actuality. An overlong burst is one in which the arrival time of the burst FINISH message is later than the actual finish time for the burst, ie. the time of the FINISH message is greater than the time of the START message for the burst plus the burst ACTT multiplied by the number of cells in the burst. A zero cell length burst represents wasted effort in the simulator, as the START has to be processed just like for any other burst. The fact that the burst actually contains no cells is only known when the FINISH arrives. Simulation objects which model a delay in the throughput of bursts (such as queues if appropriately configured) can remove zero cell length bursts if the START message for the burst has not been scheduled at the output (or can be cancelled if its scheduled time is greater than the current simulation time). An overlong burst is problematic in the sense that the FINISH message for the burst arrives later than it should. This requires the design of each simulation object, which may be required to process overlong bursts, to be capable of handling this. The information in the FINISH message will not violate the cell counts described in the START message for the burst, so any calculations made on the basis of just receiving the START message will still be correct. Of course, an object with a throughput delay will be able to resynchronise the late FINISH message to its correct time, if the error in timing is less than the delay incurred by the burst in the object.

Figure 5.9 presents the percentage of bursts to arrive at the Receiver object in each burst-level experiment which describe zero cell length bursts. As can be seen from the results, there is very little difference between the two burst-level simulator types for each set of experiments. The largest difference is between the experiments with different burst length distributions. Zero cell length bursts are typically produced when many bursts arrive, or finish, at the multiplexer input in a short space of time. If the time period to fill with cells is very small or zero (ie. an output FINISH is required at the same simulation time as the previous START), then zero cells are assigned to the output burst. In the lower burst cell length experiments, the time "density" of input START and FINISH message arrivals is greater than in the higher burst cell length experiments. This leads to the higher relative proportion of zero length bursts, although the percentage is small every time.

Figure 5.10 presents the percentage of overlong bursts which arrived at the Receiver object in each of the experiments performed. Here, the **BL2** simulator



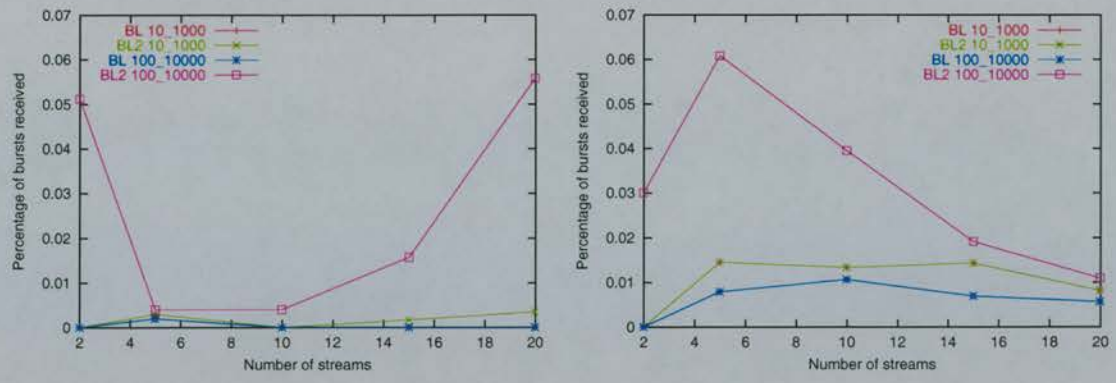
(a) "Dense" burst production



(b) "Sparse" burst production

Figure 5.9: The percentage of zero length output bursts produced by the multiplexer in the "dense" and "sparse" experiments

produces relatively more overlong bursts than the **BL** simulator, especially in the experiments involving the larger burst cell size distribution. An overlong burst can be produced when the multiplexer has used all of the cell fractions it can when allocating cells to an output burst, but more are required to fill the time period for the output burst (ie. the `FINISH` should be scheduled at a time greater than or equal to the arrival time of the input `START` or `FINISH` message which triggered the generation of the output `FINISH` message). An overlong burst can also be produced if the duration filled by the cells allocated to a burst is too small, but the maximum number of cells detailed in the `START` message has already been reached. At this point, the decision is made to schedule a late `FINISH` message for the burst, as the details already sent in the burst `START` message cannot be compromised. It is not surprising that **BL2** produces more overlong bursts as it was shown to consistently underestimate the burst ACTT in the experiments. This underestimate means that the duration of output bursts may not fill the required duration for the burst even though the maximum cell count has been provided for the burst. The greater proportion of overlong bursts for simulator **BL2** is the price to pay for the reduced burst start difference which is also a byproduct of the burst ACTT underestimation.



(a) "Dense" burst production

(b) "Sparse" burst production

Figure 5.10: The percentage of overlong (wrt. expected burst duration) output bursts produced by the multiplexer in the "dense" and "sparse" experiments

5.7 Queueing bursts

5.7.1 Introduction

The queue object has the job of controlling the transmission rates, and of applying cell loss due to buffer overflow, to cells or bursts in the cell and burst-level simulators respectively. The burst-level queue requires the techniques described in Section 3.6 to implement this behaviour. The cell-level queue can be of a much simpler design, as it can react to each and every input cell before deciding if the cell can be transmitted at the output, or lost if the queue is full. Once again, the choices made in the simulator parameters and the techniques chosen will have an influence on the accuracy and performance of the burst-level queue. For each arriving burst, the queue must calculate any cell loss incurred by the burst traversing the queue, as well as the burst ACTT of the burst when it is transmitted at the output. In the burst-level simulators, the amount of work performed by the queue is proportional to the number of component cell streams per burst, as well as to the degree of cell loss encountered. When there is little or no cell loss (ie. the minimum output ACTT for the queue is less than or equal to the burst ACTT of the input bursts) the work required per burst will be small.

The simple design (see Section 5.7.2) of the cell-level queue is also its weakness. The choice exists of either physically buffering queueing cells (whose output transmission is triggered by the queue scheduling an event for itself) or of scheduling output cells into the future as soon as they arrive. Both choices place a memory penalty on the simulator, as the *memory footprint* of the queue object is large for a large buffer size, or the number of events pending on the global event list may

be great. Many pending events will require a considerable amount of memory, as well as reducing the efficiency of the global event list primitive operations. It is obvious that a large simulation experiment with many queues could cause problems for cell-level simulation studies.

A fast and efficient queue object is essential to the burst-level simulator. This is because queues are used to police the output of multiplexers, as well as when buffering is required in the network being modelled. Policing the multiplexer output involves ensuring that the burst ACTT does not drop below a minimum specified for the communications link, as well as discarding cells if the buffer capacity for the multiplexer is exceeded. The cell-level multiplexer, on the other hand, can police its own output, without the aid of an additional queue object, due to its very much simpler design. A cell-level multiplexer is virtually identical in operation to a cell-level queue. Producing a compound multiplexer in the burst-level simulator helps reduce the complexity of both the queue and the multiplexer objects from which it is composed. To attempt to reduce the performance penalty of requiring two objects, the global event list can effectively be bypassed by making the queue object the *slave* of the burst multiplexer object. This involves the multiplexer passing events directly to the queue rather than scheduling them in the normal manner. This is the logical equivalent of integrating the queue behaviour into the multiplexer object, but without the extra complexity presented by that option. Further efficiency savings may be possible if the behaviour of the queue was integrated into the multiplexer object. However, this may lead to a very complicated implementation.

The experimental results in this section compare the accuracy and performance of the burst-level simulators with their cell-level counterparts for some simple queueing experiments. The accuracy results from the cell-level simulators are taken to be "ideal", ie. the results which the burst-level simulators should try to match. Parameters such as the number of input cell streams and the degree of cell loss encountered will be examined. The different cell level queueing object choices are also examined to show the difficulties inherent with this scale of network modelling.

5.7.2 Cell-level queueing

The fundamental behaviour of the cell-level queue is to accept cells at its input, and to ensure that the stream of cells sent from its output has an interarrival time not less than the minimum specified for the queue (ie. the minimum queue output ACTT). An input cell is discarded (ie. no output cell event is scheduled for this

cell) if the queue decides that it is fully occupied when the event carrying the cell arrives. If the queue is not full, the input cell is transmitted at the queue output port at a simulation time which represents its sojourn time in the queue. As the decision whether to discard, or to buffer, the cell for subsequent transmission can be made for every input cell arrival, the design of the cell-level object is very simple. The main choice is in how the queue actually models the buffering of cells. One option is to physically buffer the queueing cells in an *array* or *linked-list* data structure within the queue. The other choice is to immediately schedule the cell at the output upon its arrival, such that its scheduled simulation time reflects the sojourn time the cell would have had in the queue if it had been physically buffered. The two approaches are described in the following sections.

5.7.2.1 Physically buffering cells

The first option is to physically buffer each queueing cell using a data structure within the queue object. The choice of data structure chosen can be important as regards the memory footprint of the queue object. If a *dynamic* structure, such as a linked list, is used, the memory footprint of the queue object will be small if the queue occupancy is low. If, however, the queue occupancy is large, the memory footprint will be large as the pointers required for the list will need to be stored, as well as the cell objects themselves. A *static* data structure could also be used, such as an array of cells. This will give a memory footprint of a constant size, but this may be a problem for large queue sizes and many queues being modelled. If an *array of pointers* to cell objects is used, the pointer array size will be fixed, but the memory footprint will vary with the cell occupancy of the queue. One technique is to use a singly-linked list of queueing cells, such that the memory footprint varies linearly with cell occupancy in the queue.

In terms of the queue object operation itself, the queue accepts incoming `cell` events at its input, and produces output `cell` events at its output upon the receipt of `QNEXT` messages. When an incoming cell arrives, the queue checks its current occupancy and decides whether to buffer the cell or to discard it. The incoming `cell` event marks the end of the transmission of the cell, so action can be taken immediately upon receipt of the message. Some care has to be taken if the incoming cell event has a simulation time *equal* to the next output cell transmission time. If this is the case, the cell can be buffered rather than discarded (as one cell arrival and one cell departure occur concurrently). Performing this check is necessary, as the global event list is processed with strict FIFO ordering. If the input `cell` event was scheduled before, but at the same simulation time

as, the next QNEXT message for the queue, the cell will be erroneously discarded as the queue will not know that the two events happen concurrently. To avoid this, the queue keeps a record of the next output cell transmission time (ie. the scheduled simulation time of the next QNEXT message) so that it can perform this check.

When the queue is empty, the queueing of cells is governed by the interarrival times of the input cells. When the first cell arrives, it can either immediately be transmitted at the output, or some parameterised delay (to model physical passage through the queue) can be added to the output cell schedule time. A record is kept of the "last transmission time" of this cell. If the next input cell arrives at a time such that the difference between the arrival time and the last transmission time is *less than* the minimum queue output ACTT, the arriving cell is queued. This ensures that the minimum output ACTT for the queue is preserved. If the difference between the cell arrival time and the last transmission time is *greater than* the minimum output ACTT for the queue, the arriving cell can be scheduled at the output immediately (subject to any parameterised delay) as no queueing occurs.

If the queue contains cells, the mechanism for producing output `cell` events is for the queue object to schedule QNEXT messages for itself. Each QNEXT message is scheduled at the simulation time marking the *end* of the next output cell transmission. This means that the next cell to be transmitted is removed from the buffer, and scheduled at the output port, as soon as the QNEXT message generated for it arrives at the queue. After the output cell has been scheduled, the queue then checks to see if the occupancy is greater than zero, ie. whether there are any cells still buffered in the queue. If so, another QNEXT message is scheduled for a simulation time equal to the current simulation time plus one minimum queue output ACTT. This ensures that the minimum cell interarrival time between output cells is no less than the minimum output ACTT specified for the queue. If the queue contains no cells, no QNEXT message is generated and the queue waits for the next input cell to start it up again.

The technique outlined above will clearly add to the total number of events processed in a simulation experiment due to the generation of QNEXT events for every output cell. However, the advantage of this technique over the alternative described in the next section is that the number of pending events on the global event list (due to the queue operation) is kept small. The tradeoff is the faster operation of the event list primitives due to a smaller number of pending events versus the increase in the total number of events due to the use of the technique.

The memory footprint of the queue, for large cell occupancies, may also be a problem although the size of a singly-linked list of cells will be smaller than a number of event objects representing the same group of cells. This is due to the extra information which has to be carried in an event object compared to a single pointer in the singly linked list.

5.7.2.2 Forward scheduling cells

The other obvious option for the cell-level queue object operation is to immediately schedule arriving cells. This is performed such that the scheduled simulation times of the output cells reflect the sojourn time they would have had in the queue. Cell loss is determined by calculating the number of cells which would be in the queue at the arrival time of the input cell. This method requires no physical buffering of cell objects within the queue, but has the potential to produce many pending events on the global event list. This places great pressure on the event list mechanism used, as the primitive event list operations will be slowed when there are many events to deal with.

The queue operation is very simple. When an input `cell` event arrives, the arrival time (T) is compared with the "last send time" (T_{lst}) of the last cell to be sent from the queue. If T is greater than T_{lst} (ie. the arriving cell is effectively entering an empty queue), the arrival time is checked to see if it is larger than T_{lst} by at least the minimum output ACTT for the queue ($ACTT_q$). This check is required to ensure that the output cell stream has an interarrival time not smaller than the minimum queue output ACTT. If the time difference is larger than $ACTT_q$, the cell can immediately be sent at the queue output, subject to any parameterised delay specified for the queue. The value of T_{lst} is set to T to reflect the last send time of an output cell from the queue. If the time difference was less than $ACTT_q$, the queue updates T_{lst} as follows:

$$T_{lst} = T_{lst} + ACTT_q$$

The arriving cell is then scheduled at the queue output for simulation time T_{lst}, subject to any parameterised delay.

If the arrival time of an input cell is less than the last send time, this represents the fact that cells are "queueing" in the queue. To calculate the current occupancy of the queue, Equation 5.5 is used. If the current occupancy is greater than the queue maximum capacity, the cell is discarded. If the current occupancy is less than the maximum capacity, the cell can be "buffered". In reality, this involves increasing T_{lst} by the value of $ACTT_q$ and scheduling this cell at the queue output for simulation time T_{lst} (subject to any parameterised delay).

$$C_q = \left\lfloor \frac{T_{lst} - T}{ACTT_q} \right\rfloor \tag{5.5}$$

Once again, care has to be taken to ensure that a concurrent cell arrival and departure are handled correctly. This is achieved by calculating an effective "last send time" based on the current queue occupancy and the current simulation time. If T_{test} is the effective last send time, it can be calculated as follows:

$$T_{test} = T + (C_q \times ACTT_q)$$

If T_{test} is equal to T_{lst}, this indicates that an output cell has been scheduled for the same simulation time as the arrival time of the input cell. This is a concurrent arrival and departure, and thus the arriving call may be "buffered" rather than discarded.

Depending on the queue size modelled, the technique outlined above has the potential for producing a great many pending events on the global event list. If the queue occupancy is low, however, the number of pending events will be small along with the memory requirements incurred through the use of this technique. Along with the cell buffering technique described in Section 5.7.2.1, this technique will require a potentially large amount of memory when queue occupancies are high. Indeed, this method may require even *more* memory than the buffered method, as the overhead of effectively storing cells in `event` objects on the event list will be greater than storing the cell objects with a singly-linked list within the queue object. The other disadvantage of this technique is that the potentially large number of pending events, especially when several queues are modelled in one simulation, may dramatically hit the performance of the global event list primitive operations, leading to poor runtime performance of the simulation.

5.7.3 The Experiments

The simulation object structure, for the queueing experiments performed, is shown in Figure 5.11. Once again, the object structure was the same for both the cell and burst-level simulations, as were the parameters passed to each object. This allowed for a direct comparison of the results between the different simulator types. The queueing experiments differed from the multiplexing experiments in that a queue object (without consumer flow control) was inserted between the `Multiplexer` and the `Receiver` objects. The behaviour of the `RandSender`, `Multiplexer` and `Receiver` objects was the same as in the multiplexer experiments described in Section 5.6.3 on page 136. Although the cell-level multiplexer

can perform its own output cell stream queueing, a separate queue object was used so that the burst and cell-level experiments performed were equivalent. Better performance would be achievable in the cell-level simulators, if the queueing capability of the multiplexer was used instead of a queue placed at the output.

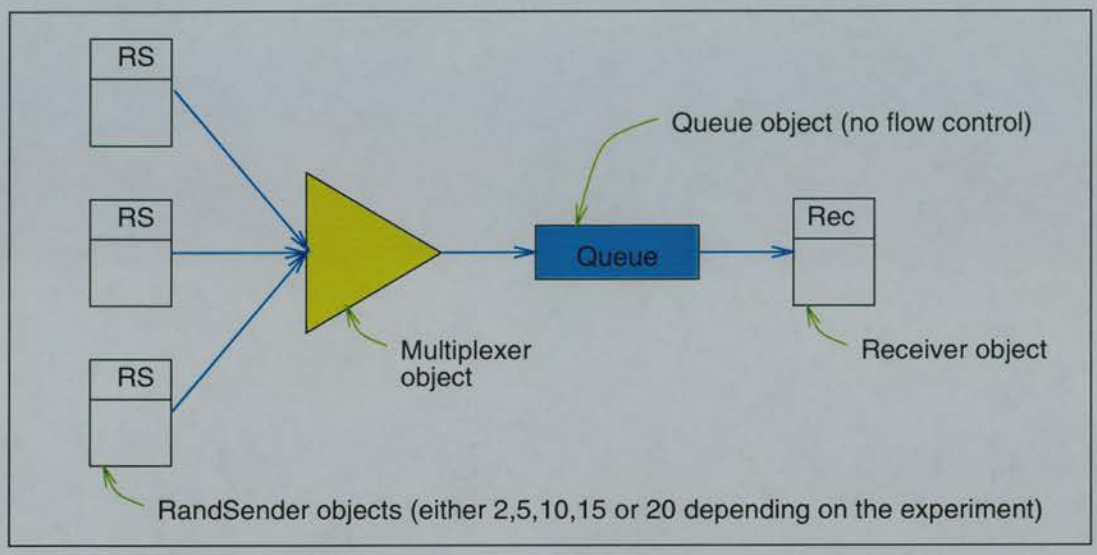


Figure 5.11: The simulator objects used in the queueing experiments

The queue object used in each simulation was a queue without flow control. The parameters used for the queue object in each simulation run, irrespective of the other parameters chosen, are shown in Table 5.9. The parameters relate to the absolute values used in the cell-level simulators, as the delay and output ACTT values need to be multiplied by 1000 in the burst-level simulators. This is due to the use of an ACTT multiplier of 1000 which is applied to every ACTT value generated in the burst-level simulators.

Parameter	Buffer size	Traversal delay	Output ACTT
Value	1000000 cells	1000	1000

Table 5.9: The queue object parameters in the queue simulation experiments.

As with the multiplexer experiments, the parameters shown in Table 5.2 on Page 139 were used for each of the simulators tested. This meant that comparative queueing results for the **BL**, **BLt**, **BL2** and **BL2t** simulators could be produced relative to the results from the **CL** and **CLt** simulators. The results produced for the "dense" and "sparse" burst arrivals in the cell-level simulators were taken to be the "ideal" results, which the burst-level simulation results were compared with.

An important choice in the cell-level simulators was which kind of queue object should be used. Either the physically buffering or the forward scheduling queue objects could be the queue without flow control in each simulation. To show the difference between the techniques, the 10 stream experiment for burst sizes ranging from 10 to 1000 cells was performed with a forward scheduling queue object and a physically buffering queue object in the **CL** simulator. The other parameters were as described for the experiments detailed above.

Table 5.10 shows the comparative performance of the experiments for the physically buffering (PB) and forward scheduling (FS) queue objects. The results follow the predictions made in the Sections 5.7.2. Although the buffering queue object requires a greater total number of events to be scheduled and processed, it has the best performance and smallest memory footprint of the two queue techniques. Even with a large event count per event index (set to 1000 to be the square root of up to 1000000 forward scheduled cell events), the performance of the forward scheduling queue is crippled by the event list handling required. Storing up to 1 million events in the global event list requires more memory than the singly-linked cell list storage for 1 million cells provided in the buffering queue. Smaller buffer sizes may favour the forward scheduling queue, but having many such queues in a simulation, when the buffer sizes are large, would clearly be impractical on all but compute servers with a great deal of physical memory.

Queue	Runtime (secs)	Ev count per ev index	Total events	Max ev pending	Memory req (MB)
PB	555	20	181027811	525	43
FS	9057	1000	141587757	1000523	67

Table 5.10: The relative performance results of a cell level simulation for a buffering and forward scheduling `Queue` object

With the performance difference shown in Table 5.10 in mind, each **CL** experiment used a `Queue` object set to physically buffer queueing cells.

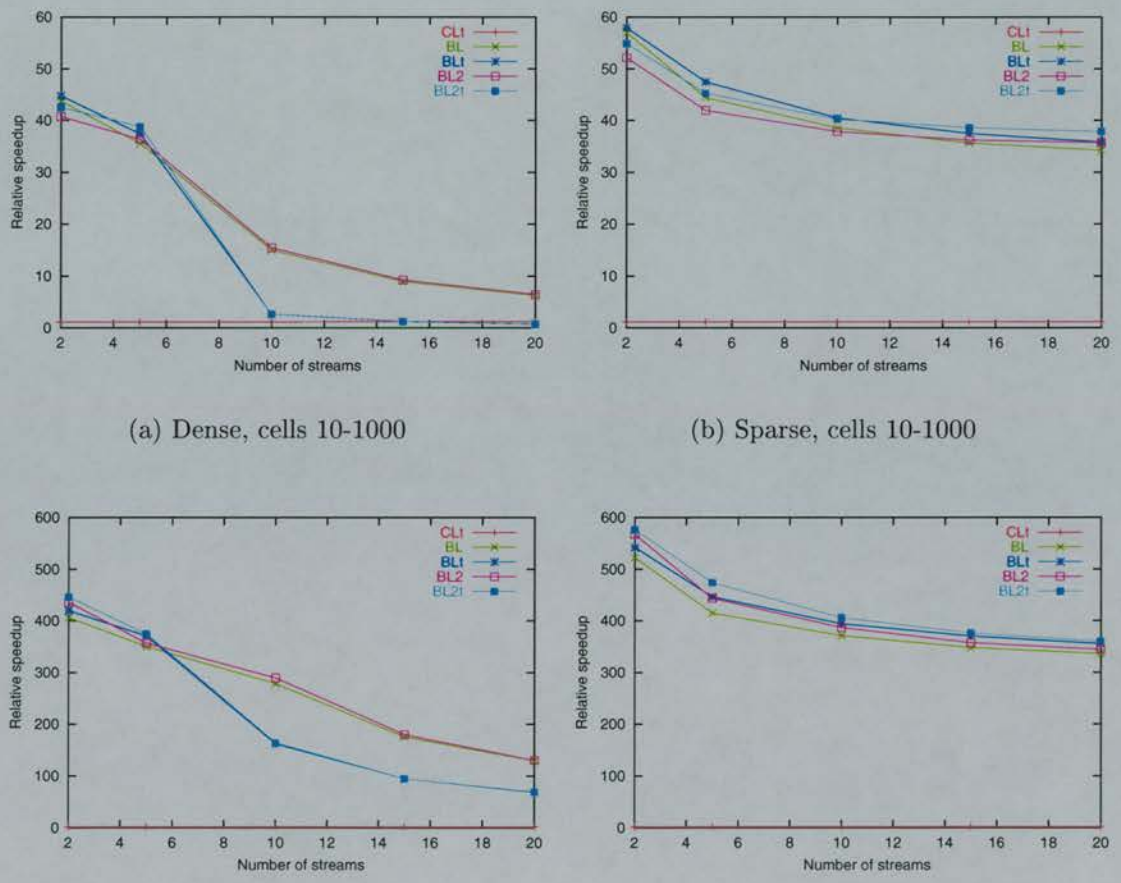
5.7.4 Results

The "dense" and "sparse" burst arrival experiments were performed in two batches, identical in the simulation parameters used. The first batch minimised the processing of burst or cell information at the `Receiver` object to give the runtime comparison results in Section 5.7.4.1. The second batch of experiments enabled all of the runtime analysis required to produce the accuracy comparisons in the sections starting from Section 5.7.4.2. Care was taken to ensure that the work-

station used for the runtime comparison results was not loaded by other user processes.

5.7.4.1 Runtime comparisons

Figure 5.12 shows the speedup obtained in each simulator tested, relative to the runtimes of the **CL** simulator. The numerical results may be seen in Table A.2 in Appendix A. As with the multiplexer experiments, the **CLt** simulator produced slightly lower runtimes than the **CL** simulator, hence the inclusion of a speedup plot. In all of the experiments performed, cell loss only occurred in the "dense" burst production experiments where the number of streams merged was 10, 15 or 20. The actual cell loss results, with a comparison between the different simulators, are presented in Section 5.7.4.3.



(a) Dense, cells 10-1000

(b) Sparse, cells 10-1000

(c) Dense, cells 100-10000

(d) Sparse, cells 100-10000

Figure 5.12: The relative runtime speedups for each simulator in the queue experiments

As can be seen from Figure 5.12, the burst-level simulators offer a speedup

relative to the cell-level equivalents. As with the multiplexer experiments, the burst size distribution used has a major effect on the speedup obtained. The larger the bursts to be multiplexed and queued, the better the performance of the burst-level simulators relative to the same experiments performed at the cell level. The burst-level simulators which use the improved integer techniques, **BL2** and **BL2t**, performed slightly better than the equivalent burst-level simulators using standard integer division. This is consistent with the multiplexer experiment results.

The most interesting observations can be made for the "dense" burst production experiment results, where the speedup offered by the burst-level simulators drops dramatically with the number of cell streams to be merged. The effect is particularly pronounced in the **BLt** and **BL2t** simulators which use a post-ordered tree global event list. Table 5.11 shows the total event counts for each of the simulation experiments where cell loss was experienced. Also shown are the maximum number of events pending on the event list, as well as the total number of events cancelled from the event list by the `Queue` object. The `Queue` object schedules burst `START` and `FINISH` messages into the future in order to simulate ACTT changes due to the burst queueing (as described in Chapter 3). These messages may need to be cancelled if the input `FINISH` message for that particular burst means that different output `START` and `FINISH` messages need to be scheduled. The degree of burst scheduling and subsequent cancellation represents wasted effort, which will reduce runtime performance. However, as is shown in Table 5.11, the proportion of cancelled events relative to the total number of events executed was very small in each experiment.

The effort required to remove an event from the global event list depends on the event list algorithm used. In the indexed event list, event removal can require as little as two pointer reassignments, as long as the event is not pointed to by an event index. In that case, the event index needs to be reassigned to an event either preceding, or succeeding, the event to be removed. In the post-ordered tree, event removal depends on whether the event to be removed has no branches, a right branch only, a left branch only or both branches. The least effort is required in the "no branch" case which requires one pointer reassignment. The "left branch only" or "right branch only" cases require reassigning the branch to the preceding event of the event to be removed. The most effort is required when the event to be removed has both branches. Rather than recursively reassign the events in the branches, the event to be be removed is set to be a `NULL` event with a simulation time equal to the event simulation time. The `NULL` event is not processed when it

"Dense" burst production, burst size 10 to 1000 cells					
	CL	BL		BL2	
Streams	Total Events	Total Events	Cancelled Events	Total Events	Cancelled Events
10	181027811 (525)	1097952 (10779)	867	1098042 (10781)	826
15	232106315 (782)	1646329 (16117)	1683	1646445 (16126)	1625
20	283010024 (1042)	2194390 (21436)	2655	2194356 (21461)	2672
"Dense" burst production, burst size 100 to 10000 cells					
	CL	BL		BL2	
Streams	Total Events	Total Events	Cancelled Events	Total Events	Cancelled Events
10	1791141720 (525)	1099554 (1229)	100	1099480 (1229)	149
15	2301888803 (785)	1649363 (1851)	179	1649141 (1849)	296
20	2810743125 (1042)	2199164 (2378)	283	2198756 (2376)	486

Table 5.11: The total number of events (with max. no. of pending events in brackets) processed, and the number of events cancelled by the `Queue` object, for the "dense" queueing experiments.

becomes the "current" event. However, keeping the `NULL` event in the tree ensures that new events added to the tree are inserted in the correct place relative to their scheduled simulation time.

As can be seen from Table 5.11, the burst-level simulators have large maximum numbers of pending events in the "dense" experiments where the burst size varies from 10 to 1000 cells. This is coupled with a burst cancellation count which increases with the number of component streams simulated. As described in Section 3.2.4.1, for an event list containing N events, the worst case time complexity of event insertion into the list is $O(\sqrt{N})$, if the number of events per event index is $\sqrt{N}$. The value used in the **BL** and **BL2** experiments for the number of events per event index was 20. Clearly, the value 20 is too low when compared with the maximum numbers of pending events shown in Table 5.11. The "dense" burst production experiments for the burst size range of 10 to 1000 cells were performed again, this time with the event index event count set to 100. The difference in runtime speed up over the same experiments in the **CL** simulator are shown in Figure 5.13. With a larger number of events represented per event index, the burst-level simulators achieve better performance. This helps illustrate the hard

choices modellers have as to the parameters they must set in their simulator to maximise efficiency. As can be seen from Figure 5.13, the higher event count per event index actually *reduces* performance, when the number of pending events is small. Choosing an optimal value is a difficult task, unless one can use feedback from previous simulation results where similar parameters were used. Some estimate of the maximum number of pending events needs to be made which will depend on the number, and type, of objects in the simulation. When subsequent experiments are performed to improve the confidence intervals of the results, the results from the first run can be used to "fine tune" the simulation event list parameters.

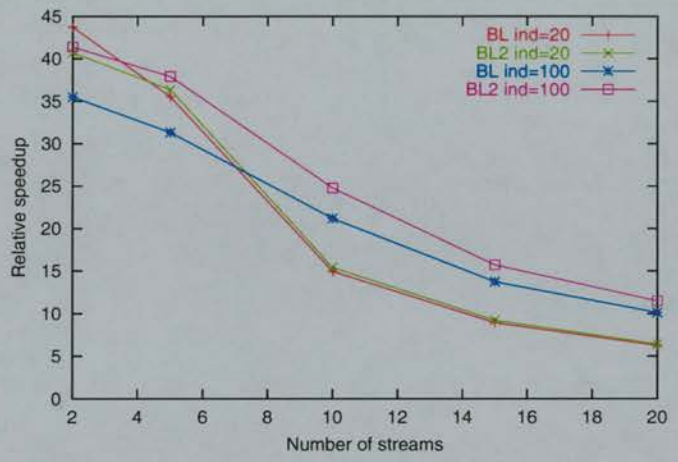


Figure 5.13: Runtime improvement for **BL** and **BL2** when the event index event count was increased from 20 to 100 in the "dense" burst production experiments (burst size range 10-1000 cells).

The poor performance of the post-ordered tree global event list burst-level simulators is obvious, especially in Figure 5.12(a). As described in Section 3.2.4.1, the performance of the post-ordered tree event insertion algorithm is dependent on the distribution of the simulation times of the events inserted. To help ascertain why the performance was so poor, the **BLt** simulator was compiled to use the *GProf* [48] runtime code execution profiling tool. The "dense" burst production experiment for 10 streams with burst sizes varying between 10 and 1000 cells was then rerun. Upon examination of the runtime execution trace, it was found that the simulator spent approximately 93% of its execution time in the `add` method of the `ev_list` event list class. This shows that the distribution of simulation times for events to be added to the event tree in these simulations made the event list perform poorly. When the distribution is "friendlier" to the post-ordered tree algorithm, such as in the multiplexer experiments in Section 5.6, better

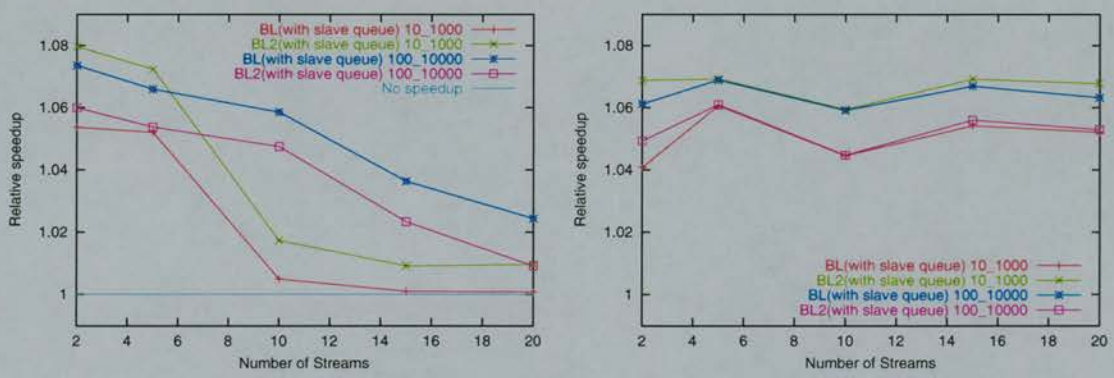
performance can be obtained than for a global event list based on the indexed event algorithm. However, the post-ordered tree based cell-level simulator, **CLt**, still offered better performance than the indexed event list alternative. As each cell is simulated individually, the distribution of simulation times for events to insert into the global event tree will be very different from those in the burst-level simulations.

The performance of the burst-level simulators is more consistent in the "sparse" burst production experiments (Figures 5.12(b) and 5.12(d)). This is as would be expected, due to the amount of work required in the multiplexer and queue objects for these burst distributions. As was shown in the multiplexer experiments, the low number of cell streams per burst produced in the "sparse" experiments aided the runtime performance. When there is no cell loss, the `Queue` object does not have to perform burst splitting to allocate cell losses among the component streams in a burst. The queue will still have to police the output burst ACTT for each burst, applying any changes to the component stream ACTTs carried in the burst. Accounting for ACTT changes in the output bursts will also be required, ie. producing pairs of `START` and `FINISH` messages to denote burst ACTT changes for each burst.

When the `Queue` object has to perform cell loss, more work is required to perform the allocation of the cell loss among the component streams of a burst and this will impact performance. The multiplexer experiments showed that the "dense" experiments produced many bursts, whose cell counts are related to the original burst size distributions for the source `RandSender` objects. The greater the average number of cells in each multiplexer input burst, the greater the average burst size of the multiplexer output bursts. The "dense" experiments make the multiplexer perform many burst ACTT calculations and burst splits to produce the many output burst messages produced. The work performed by the `Queue` object is proportional to the number of input bursts processed, and so the "dense" experiments require more processing effort, especially when cell loss must be calculated. This helps explain the decreasing performance of the burst-level simulators with increasing number of input cell streams in the "dense" experiments. The speedup is better in the "dense" experiments where the source burst size ranges from 100 to 10000 cells, as each multiplexer output burst describes, on average, a greater number of cells than in the case of burst sizes from 10 to 1000 cells.

One means of helping to improve the runtime performance of the burst-level simulators is to make the `Queue` object the slave of the `Multiplexer` object. This

means that output bursts from the multiplexer are not scheduled on the global event list, but instead passed directly to the `Queue` object. As the multiplexer ensures that it produces its output `START` and `FINISH` messages in strict order of increasing simulation time, the queue can set its simulation time to be that of the burst being passed to it. This allows the queue to correctly schedule output burst events on the global event list. To gauge the level of any runtime improvement offered by this technique, the burst-level simulations were performed again, this time with the `Queue` object made to be a slave of the `Multiplexer` object. The experiment runtimes were collected, and compared to the runtimes of the first set of experiments. Figure 5.14 shows the relative runtime improvement of the event list bypass experiments, compared with the same burst-level experiments which keep the `Queue` and `Multiplexer` objects separate. The "dense" experiments were performed with a maximum event count of 20 for each event index.



(a) Dense burst production

(b) Sparse burst production

Figure 5.14: The relative runtime speedups obtained when the `Multiplexer` bypassed the global event list to drive the `Queue` object for the **BL**, **BL2**, **BLt** and **BL2t** simulators.

As can be seen from the results in Figure 5.14, bypassing the event list to join the `Multiplexer` and `Queue` objects offers a small improvement in the experiment runtimes for each of the burst-level simulators. The improvement is almost constant, relative to cell stream count, in the "sparse" experiments. The situation is different in the "dense" experiments, especially when cell loss is encountered in the 10, 15 and 20 cell stream cases. This shows that the work required to perform the cell multiplexing, queueing and cell loss in these experiments dominates the experiment runtimes. The general performance decreases with an increasing number of cell streams, but bypassing the event list still helps produce an overall decrease in runtime. The cases where the burst size is between 10 and 1000 cells

are crippled by the small event count per event index. However, the event list bypass still offers a performance improvement, which would still be apparent if a larger event count per event index were used.

5.7.4.2 The accuracy comparison results

In the second batch of queueing experiments, the `Receiver` object was set to record the details of each burst received. The data recorded, in both the cell and burst-level simulations, was the same as was recorded in the multiplexing experiments described in Section 5.6.4.2 on page 145. Comparisons between the burst and cell-level queueing experiment results could then be made. Once again, the results from the experiments using a post-ordered tree event list (ie. **CLt**, **BLt** and **BL2t**) were identical to the results obtained from their indexed event list counterparts, and are thus not discussed.

5.7.4.3 Cell loss comparisons

The first analysis to be performed was that of the relative cell losses encountered in the burst and cell-level simulation runs. Cell loss was only encountered in the "dense" burst production experiments when the number of simulated cell streams was 10, 15 or 20. Table 5.12 shows the total percentage of incoming cells discarded by the queue object in each of the experiments where cell loss was encountered. As can be seen from the results, the cell loss increased with the number of streams simulated, but each simulator type was in good agreement as to the overall proportion of cells lost by the queue. To analyse the cell loss further, the cell totals recorded by the `Receiver` object for each simulated cell stream were compared. Table 5.13 presents the average difference (in cells) between the burst-level experiment stream totals and the cell-level experiment stream totals (as recorded for simulator **CL**). At the "stream scale" the different simulators are still in reasonable agreement. The average difference in the number of cells received for each stream is close to zero in all cases. The values of the standard deviation of stream cell count differences are almost identical in both types of burst-level simulator used. The magnitude of the standard deviation, when compared to the average number of cells per stream in the cell-level results, is small.

To examine the comparative cell loss at the burst level, the numbers of cells received for each burst in each simulation run were then compared. This analysis compared the cell counts for *all* of the bursts received by the `Receiver` object, to give an overview of the effects of the different burst-level simulators on the accuracy of the cell loss encountered for all of the simulated bursts. The cell

"Dense" burst production, burst size 10-1000			
Streams	CL	BL	BL2
10	21.97	21.97	21.97
15	47.91	47.91	47.91
20	60.90	60.90	60.90
"Dense" burst production, burst size 100-10000			
Streams	CL	BL	BL2
10	23.75	23.75	23.75
15	49.10	49.10	49.10
20	61.79	61.79	61.79

Table 5.12: The total percentage of input cells lost in the queue in the experiments where cell loss occurred

counts recorded for each burst received in the **CL** experiments were regarded as the "ideal" to be emulated by the burst-level experiments. For each set of burst-level simulation results, the cell counts for each burst received by the `Receiver` object were compared with the cell counts for the same burst (as identified by the burst unique identifier) in the equivalent cell-level simulation experiment. The difference between the two counts was recorded to produce an average difference, with standard deviation, in cell loss per burst between the burst and cell-level simulators. The average burst cell count difference results, per received burst, are presented in Table 5.14. The standard deviation of each average burst size difference is also given, both as a cell count and as a percentage of the average number of cells per burst (as recorded in the **CL** simulation results for each experiment).

As can be seen from Table 5.14, both burst-level simulators produce pretty near identical results as regards the average cell count difference to the cell-level results per burst. The average final burst size difference, in cells, is close to zero in every case, but the standard deviation increases with the number of cell streams. At its worst, the standard deviation can be around 10% of the mean burst size, which shows fairly wide variation. This means the accuracy of the burst-level queueing algorithms decreases, at the burst scale, when heavy cell loss is encountered. The variation in the accuracy of the burst-level techniques is much lower when the results are examined at the stream and "total simulation run" scales.

Greater variation of burst size difference is seen in the "dense" burst production experiments where the initial burst size can vary between 10 and 1000 cells. As described in the multiplexer experiment result analysis, the multiplexer produces many output bursts, each describing a small number of cells in this case. When the queue object has to process these bursts, and perform any calculated

Dense burst production, burst size 10-1000			
	CL	BL	BL2
Streams	Mean cells per stream	Mean cell count diff per stream	Mean cell count diff per stream
10	3944031	8.19 (2164.03)	1.40 (2117.72)
15	2631696	13.00 (2582.07)	0.60 (2540.35)
20	1974071	8.90 (1841.01)	0.70 (1835.67)
Dense burst production, burst size 100-10000			
	CL	BL	BL2
Streams	Mean cells per stream	Mean cell count diff per stream	Mean cell count diff per stream
10	38540211	8.69 (18395.41)	0.5 (18415.02)
15	25716869	25.33 (18431.48)	0.13 (18384.73)
20	19290920	25.65 (16265.12)	0.5 (16260.05)

Table 5.13: Mean cell count difference per cell stream, at the `Receiver` object, between the burst and cell-level results. The mean number of cells received per stream for the **CL** results, as well as the standard deviation of each mean difference, are also shown.

cell loss, the burst splitting technique used to apportion the total loss for the burst over the component streams will judge each burst in isolation. If each burst is small, and cell fractions have been used to fill the cell counts, the scope for error when applying cell loss is magnified. As can be seen in the variation of the difference around a mean difference of almost zero cells, the burst-level queueing algorithms both *overestimate* and *underestimate* the queueing cell loss for bursts. The degree of this variability is linked to the degree of total cell loss encountered, the number of streams described by each queueing burst and the size (in cells) of each queueing burst. The larger the average size of each queueing burst, the more accurate any cell loss calculation will be due to errors "averaging out" over the greater number of cells. This is seen to some degree in the smaller variations seen in the "dense" experiments where the initial burst sizes were 100 to 10000 cells.

The results suggest that if detailed burst sizes are needed on a "per burst" basis in a simulation (as could happen in the simulation of communication protocols), the burst-level techniques will show large variability if heavy cell loss is encountered. If the burst sizes are large, and the degree of cell loss is small, the

Dense burst production, burst size 10-1000 cells							
	CL	BL			BL2		
No. str	Av cells per burst	Av cells	Mean diff	sd	Av cells	Mean diff	sd
10	394.40	394.40	0.21	18.34 (4.6%)	394.40	0.22	18.33 (4.6%)
15	263.17	263.17	0.05	22.65 (8.6%)	263.17	0.05	22.65 (8.6%)
20	197.41	197.41	0.04	20.55 (10.4%)	197.41	0.04	20.55 (10.4%)
Dense burst production, burst size 100-10000 cells							
	CL	BL			BL2		
No. str	Av cells per burst	Av cells	Mean diff	sd	Av cells	Mean diff	sd
10	3854.02	3854.02	1.13	179.31 (4.6%)	3854.02	1.14	179.24 (4.6%)
15	2571.69	2571.69	-0.35	207.56 (8.1%)	2571.69	-0.36	207.51 (8.1%)
20	1929.07	1929.07	-0.12	171.42 (8.9%)	1929.07	-0.13	171.35 (8.9%)

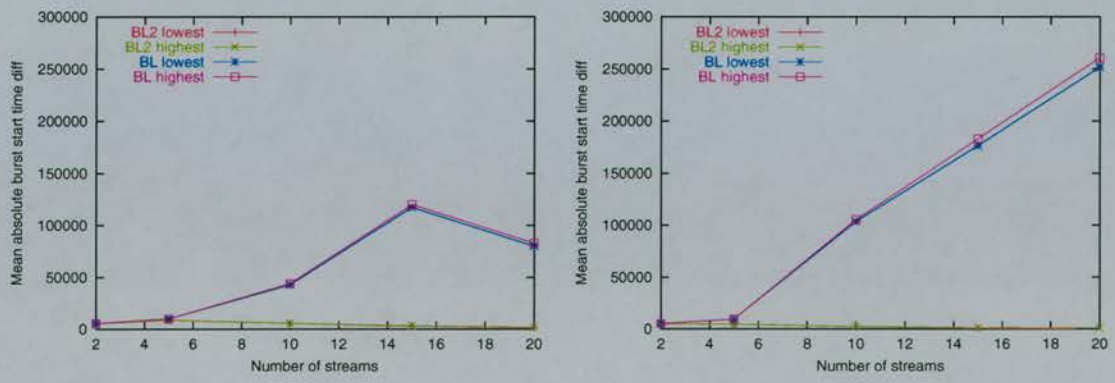
Table 5.14: Average number of cells per burst in each burst-level simulation where cell loss was encountered. The average (with standard deviation) of each final burst size in cells, cf. the equivalent final burst size in the **CL** results, is also presented. In brackets are the standard deviations of the mean burst size difference expressed as a percentage of the mean burst size.

burst-level algorithms will give better results on a per burst basis. If, however, the object of a simulation is to determine average cell losses over broader time scales (as could occur when using simulation to help dimension broadband backbone networks) the burst-level queueing techniques offer very similar results to the cell-level simulator, when considering total cell loss at a network element, even for heavy cell loss.

5.7.4.4 Burst start time comparison

The next analysis performed was to compare the burst start times, as recorded by each `Receiver` object, for each burst in equivalent cell and burst-level simulation runs. As in the multiplexer experiments, the mean absolute time difference between the burst-level and cell-level burst start times was computed. As this analysis was performed for the bursts received for each stream, the lowest and highest mean differences per experiment set were recorded. An experiment set comprised the results for 2, 5, 10, 15 and 20 streams, for either "dense" or "sparse" burst production, for both burst size ranges (ie. 10-1000 and 100-10000). The range of

mean burst start time differences between the burst and cell-level simulators for the "dense" burst production experiments can be seen in Figure 5.15. The actual data for these experiments may be seen in Table A.3 in Appendix A.



(a) No. cells per burst 10-1000

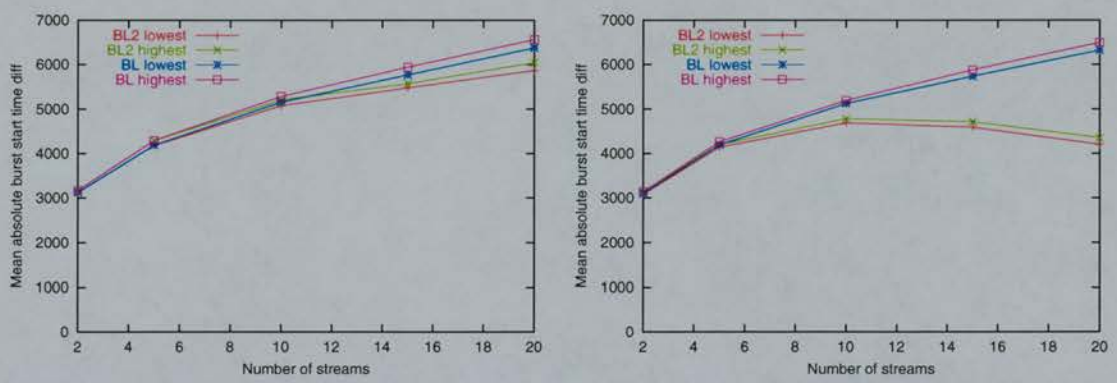
(b) No. cells per burst 100-10000

Figure 5.15: Lowest and highest mean absolute burst start time differences for the burst-level simulations in the "dense" burst production experiments

The results for the average burst start delay shown in Figure 5.15 are entirely consistent with those from the multiplexing experiments. The **BL2** experiments once again show a dramatically smaller average delay for burst start times when compared with the same experiments performed with the **BL** simulator. The underestimate of the burst ACTT by the multiplexer in the **BL2** simulator leads to the multiplexer suffering smaller cumulative burst message delays at its output. As the `Queue` object directly receives the burst messages generated at the multiplexer output, it too benefits from the lesser delay incurred in the **BL2** multiplexer. The queue will manipulate burst ACTT values to simulate the queueing of cells, but it relies on the timings of the input `START` and `FINISH` messages, as well as the value of the burst ACTT, to synchronise its actions. The queue may also incur delay in sequencing output `START` and `FINISH` messages (as the strictly sequential time order cannot be compromised) but the multiplexer is the major influence in determining the extent of the delay, due to it calculating the burst ACTT.

The results for the "sparse" burst generation experiments also mirror those seen in the multiplexer experiments. The range of average burst start delays for the burst-level simulations, compared to the results of the equivalent **CL** simulations, are shown in Figure 5.16. The raw results may be seen in Table A.4 in Appendix A. In every experiment, the **BL2** simulator produces a smaller average burst start delay when compared to the same experiments performed with

the **BL** simulator. The magnitude of the delays is also very much smaller than that experienced in the "dense" **BL** experiments. Of interest is the increasing difference between the **BL** and **BL2** average burst start delays with increasing stream count in the "sparse" experiments when the burst size range is 100-10000 cells. This suggests that the underestimate of the burst ACTT by the multiplexer, coupled with bursts queueing in the `Queue`, helps to reduce the average burst start time delay. This occurs because queueing bursts can "recover" any delay incurred at the multiplexer by waiting behind other bursts in the queue. As there is no cell loss in the "sparse" experiments, the burst start time will be much closer to those recorded in the cell-level simulations, as the number of cells per burst is identical. The same delay recovery effect is seen to a lesser extent in the "sparse" results when the burst size range is 10-1000 cells. Larger burst sizes mean that a greater number of bursts will be queued, thus allowing a greater likelihood of start time delay recovery while the bursts are waiting in the queue.



(a) No. cells per burst 10-1000

(b) No. cells per burst 100-10000

Figure 5.16: Lowest and highest mean absolute burst start time differences for the burst-level experiments in the "sparse" burst production experiments

5.7.4.5 Burst ACTT comparison

The next part of the analysis was to compare the final burst ACTT values, as recorded by the `Receiver` object, for the burst and cell-level simulators. As in the multiplexer experiments, the ACTT values for every burst received in each experiment were compared, rather than perform the analysis on each cell stream individually. This was done to give a general overview of the effect of burst queueing on all of the bursts simulated in each experiment. For each unique burst (as identified by the unique burst identifier) an ACTT ratio was defined to be the ACTT value from the burst-level experiment divided by the ACTT value

from the equivalent cell-level experiment. The average final burst ACTT ratios for each of the burst-level simulators, when compared to the cell level results, can be seen in Figure 5.17. The raw data for the experiment results can be seen in Table A.5 in Appendix A.

As was found in the multiplexer experiments, there is little difference between the mean ACTT ratio values obtained in the **BL** and **BL2** simulation results. However, unlike the multiplexer experiments, the ACTT ratios obtained in the queueing experiments are mainly *overestimates* rather than underestimates. This is particularly noticeable in the "dense" burst production experiments, as seen in Figure 5.17. The cause of this overestimate is related to several factors.

Firstly, the final cell counts per burst in the burst-level simulations are variable with respect to the cell-level results. This variation increases with decreasing burst size and increasing number of component streams, and is reflected in the increasing overestimate of the ACTT ratio with an increasing number of streams.

Secondly, as part of the queueing process, the queue has to restrict burst ACTT values to the minimum allowed at its output (ie. the queue $ACTT_{min}$ value). In turn, each component ACTT value in the burst is scaled to take account of the new burst ACTT for the burst (as described in Section 3.6 in Chapter 3). This process will introduce inaccuracy into the ACTT values for each component stream of each burst. This inaccuracy will decrease with larger burst sizes, as any error is averaged over the number of cells in each stream. For large burst sizes, each multiplexed burst is a better representation of the cell stream mix after multiplexing. The relative reduction of the ACTT ratio overestimate in Figure 5.17(b) (when the burst size range is 100-10000 cells) helps demonstrate the improvement in accuracy with larger burst sizes.

Thirdly, the allocation of cell loss amongst the component streams in a burst will influence the final ACTT value for each component. As the cell stream composition of each burst is an abstraction, the cell loss per component becomes more critical as the burst size decreases. Cell loss from small bursts will lead to overestimation of the ACTT for each stream, as the duration of each cell stream within the burst is less likely to match the duration of the *entire* burst. When the burst size is large, the cell loss calculation for each component becomes more accurate, as each burst is a better representation of the multiplexed cell streams it describes. If a final burst consists of many short burst fragments (as is the case in the "dense" experiments where the burst size range is 10-1000 cells), the net result is an ACTT overestimate. This is another contributory reason as to why the ACTT ratio overestimate decreases with increasing burst size, as shown in

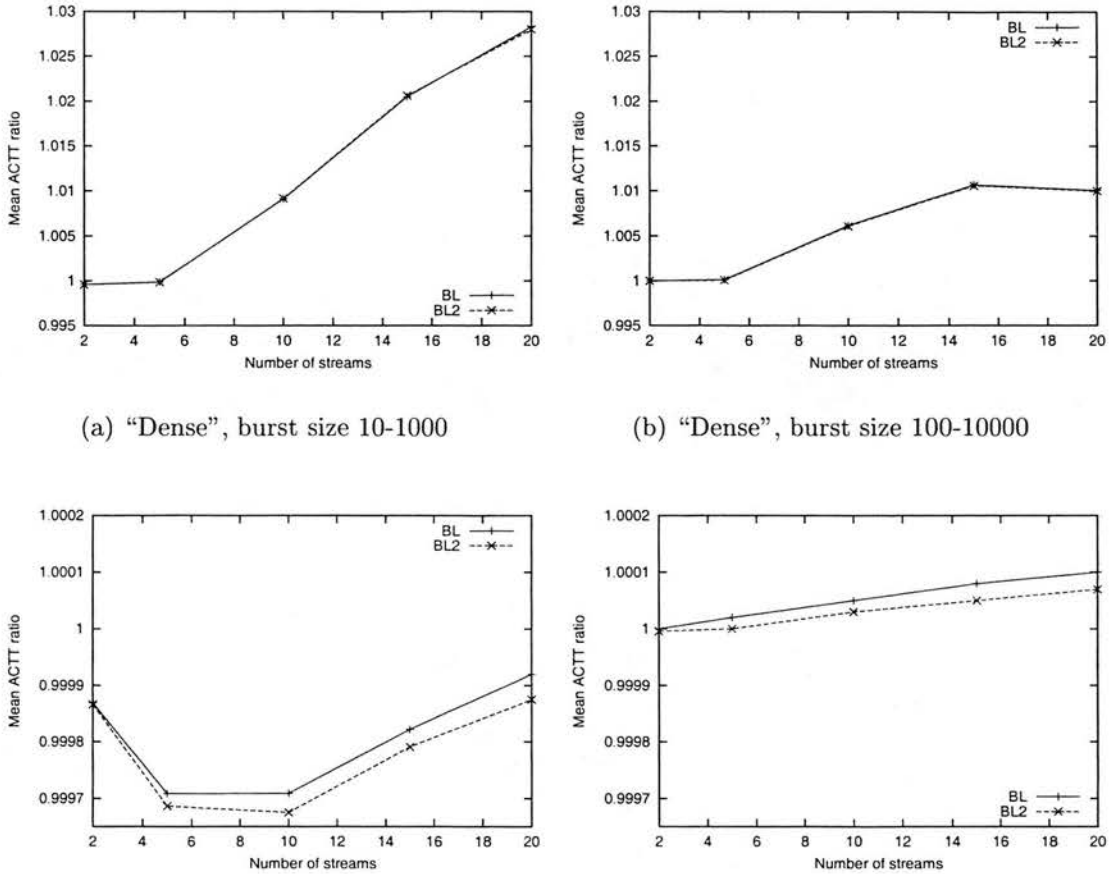
Figure 5.17(b). The cell-level simulator is more precise regarding cell loss, as it can react to each and every cell arriving at the queue. The burst-level simulators have to apply cell loss to bursts which are already abstractions of cell multiplexing behaviour. Any averaging effect on errors in the burst abstraction improves with increasing burst cell size.

The effect of queueing on the final average burst ACTT ratios is much smaller in the "sparse" burst production experiments. The mean ACTT ratios in the small burst size range case (Figure 5.17(c)) are an underestimate, while for the larger burst size range (Figure 5.17(d)), the net effect is a slight overestimate. This is as would be expected, as burst queueing is more likely in the larger burst size case. No "sparse" experiments suffer cell loss, but burst delay due to queueing is still a factor. This indicates that the rescaling of burst ACTT values by the queue, in order to preserve a minimum output ACTT, adds a slight overestimate to the final average burst ACTT ratios recorded for each cell stream. The average ACTT ratios for the **BL2** results are slightly smaller than the average ACTT ratios for the equivalent **BL** results. This is due to the burst ACTT underestimate introduced by the multiplexer helping to reduce the scale of the overestimate added by the queue.

5.7.4.6 Zero and overlong burst comparisons

The final analysis performed on the queueing experiment results was to determine the numbers of zero cell length and overlong bursts produced by the queue. A zero cell length burst is defined as a pair of `START` and `FINISH` messages which describe a possibly non-zero timespan which contains no cells (ie. the `START` message will have a non-zero total cell count, but the accompanying `FINISH` has a zero total cell count). An overlong burst is one in which the arrival time of the `FINISH` message is greater than the time of the `START` message plus the product of the cell count and the burst ACTT (ie. the actual duration of the burst of cells as described by the `FINISH` message).

Figure 5.18 presents the percentage of zero bursts received by the `Receiver` object in each of the "sparse" burst-level experiments. The "dense" zero cell length bursts are not shown, as all of the experiments produced no zero cell length bursts apart from the 5 stream case where the percentage was 0.004%. As can be seen from the figure, the percentage of zero cell length bursts is very small. This is mainly due to the parameterised delay set in the queue. If the queue detects that an input burst is of zero cell length, it can either ignore the burst entirely if an output `START` for that burst has not been sent, or cancel the



(a) "Dense", burst size 10-1000

(b) "Dense", burst size 100-10000

(c) "Sparse", burst size 10-1000

(d) "Sparse", burst size 100-10000

Figure 5.17: Mean ACTT ratios for all of the bursts received at the `Receiver` object in the queue experiments

previously scheduled output `START` for the burst. This is possible if the scheduled simulation time of the output `START` is *greater* than the current simulation time. The queue holds a pointer to the last output `START` message generated, such that the removal of the message from the global event list is straightforward. The size of the parameterised queue delay helps to determine the percentage of zero cell length bursts which can be cancelled at the queue output. The results shown in Figure 5.18 indicate that both burst-level techniques benefit from the delay in the queue for removing zero cell length bursts. When queueing occurs with no cell loss, as in the "sparse" experiments, burst `START` and `FINISH` messages are generated to denote ACTT changes in the output bursts. This action has the potential for generating zero cell length bursts which cannot be recovered if their scheduled times are less than the sum of the current simulation time plus the output queue delay.

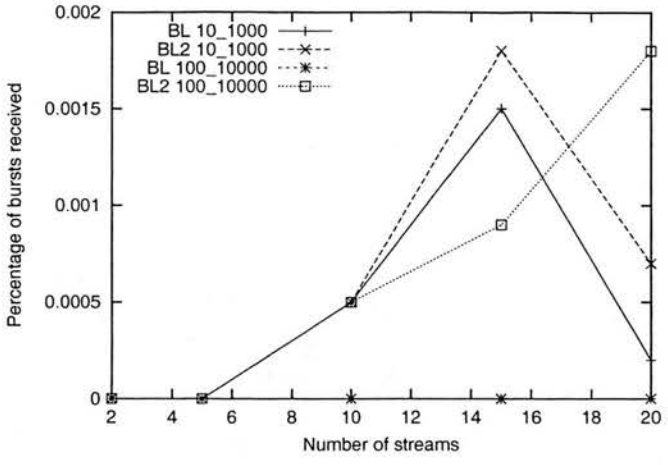


Figure 5.18: The percentage of zero cell length output bursts produced by the queue in the "sparse" experiments

The percentage of overlong bursts received by the `Receiver` object in each of the queueing experiments was found to be very low. Only the **BL2** results showed any significant number of overlong bursts in the experiments where the burst size range was 100-10000 cells. The percentage of overlong bursts was very small, and the results are shown in Table 5.15.

Streams	2	5	10	15	20
BL2 "Dense"	0.2539	0.0285	0.0010	0.0007	0.0005
BL2 "Sparse"	0.1304	0.1591	0.0885	0.0402	0.0268

Table 5.15: The percentage of overlong bursts received by the `Receiver` object in the **BL2** simulations when the burst size range was 100-10000 cells.

Overlong bursts are produced when the queue is instructed to schedule an output `FINISH` message for a burst *after* the simulation time calculated for the message to be scheduled at. This can indicate a mismatch between the arrival time of the input `FINISH` message for the burst, and the calculated queue empty time. Such a mismatch can occur when the input burst is itself an overlong burst (as produced by the multiplexer). The parameterised queue delay can help to minimise this mismatch, depending on its size.

An overlong burst can also be produced when the burst is split to simulate an output ACTT change as the burst is queueing. This happens when a burst with an ACTT larger than the minimum queue output ACTT queues behind another burst already in the queue. Some of the cells in the new burst will queue themselves (in the time taken to empty the previous burst from the queue) and so the queued portion of the new burst must be sent at the minimum queue ACTT. The remainder of the new burst is sent at the original input ACTT of the burst

(ie. as these cells will have not queued). If the burst ACTT is an underestimate, the queue can assign too many cells to be sent in the portion sent at the minimum output ACTT of the queue. If there is no cell loss in the queue, the arrival of the `FINISH` message for the burst will be at a simulation time *after* the calculated queue empty time for the remainder of the cells sent at the original burst input ACTT. If this difference cannot be recovered with the parameterised queue delay, the queue has no option but to send the final `FINISH` message for the burst later than the time it calculated the message to be scheduled for. This produces an overlong burst. The larger the burst size, the greater the likelihood of this method of overlong burst production, especially if there is no cell loss. The results for the **BL2** simulator when the burst size range is 100-10000 cells in Table 5.15 support this. As was seen in the multiplexer experiments, the degree of burst ACTT underestimation in the **BL2** multiplexer decreases with increasing stream count, and this is supported by the decreasing number of overlong bursts with increasing stream count in the results.

Very few overlong bursts are produced in the experiments where cell loss is encountered. This is because queueing with associated cell loss alters both the cell count and the burst ACTT for a burst. The queue will control the generation of the burst `START` and `FINISH` messages, which will be scheduled later than the original arrival times of the input messages for the burst (ie. as the burst will have queued). In this case, the timings of the output `START` and `FINISH` messages will match the calculated times exactly and no overlong bursts will be produced.

5.8 Demultiplexing bursts

5.8.1 Introduction

The cell and burst-level demultiplexers are the functional inverse of their multiplexer counterparts. The demultiplexer has the job of accepting a single stream of bursts or cells (depending on the simulator used) and routing these to its output ports, depending on some destination address mapping. The situation is complicated in the burst-level demultiplexer, as each input burst may contain one or more individual cell streams, each of which may be routed to any of the output ports. For an arriving burst, the demultiplexer must sequentially examine the individual cell streams carried and then route them to its output ports. The routing is based on some destination address mapping for each stream. Once an incoming burst has been processed, each output port must produce suitable `START` and `FINISH` messages describing the one, or more, streams from the input

burst which will be sent from that port. This function requires the demultiplexer to have some multiplexing capability, so that it can merge cell streams into burst message pairs at each output port. Cell buffering in the burst-level demultiplexer is implemented with a compound object that places a queue object on each demultiplexer output port. The queue performs any necessary output rate policing and cell loss due to buffer overflow. The overhead of requiring a compound burst-level demultiplexer can be reduced by making each output queue object a *slave* to the demultiplexer object. This bypasses the global event list for each transfer of a burst `START` or `FINISH` message from the demultiplexer to each output queue. This is only possible when the output queues do not use flow control with the objects connected to their output ports.

The cell-level demultiplexer can have a very much simpler design than the burst-level variant. The demultiplexer receives a stream of cells at its input which have a minimum interarrival time decided by some other object "upstream" of the demultiplexer. Upon receipt of an input `cell` message, the cell-level demultiplexer can route the cell to one of its output ports based on the destination address mapping for each cell. As the cell-level demultiplexer can react to each and every incoming cell, each output port can be thought of as a cell-level queue. With this in mind, the demultiplexer can perform its own output port cell rate policing and any buffer overflow cell loss in the same way as the cell-level queue. This, of course, presents the same problems as found in cell-level queueing, namely whether to physically buffer or forward schedule queueing cells.

The cell-level demultiplexer is effectively a set of parallel cell-level queues which are fed from a single input port. In the demultiplexer implementation each queue is not a separate object, but the functionality programmed for each output port is the same as if an individual queue object were being implemented, ie. the behaviour would be exactly the same if a separate queue object were used. When an input `cell` message arrives, the output port mapping table is examined to see which output port the cell should be routed to. The port mapping table can be conveniently indexed by the destination network address carried in each `cell` object. Other mappings are possible and can be easily implemented in the cell level demultiplexer. Once the output port for the incoming cell has been decided upon, the cell is "buffered" by the queue representing that output port. The queueing strategy for each output port can be either of the techniques outlined in Section 5.7.2, namely physically buffering or forward scheduling cells at each output port. A demultiplexer with many output ports, and a large buffer capacity for each port, would require a large memory footprint in a simulation. However,

physically buffering cells with an efficient singly-linked list structure will use less memory than forward scheduling each cell for each output port.

5.8.2 The Experiments

Figure 5.19 shows the object structure for each of the demultiplexing experiments performed. To provide direct comparison between the burst and cell-level simulators, the same experiments, with the same simulation object structures and parameters, were performed using each cell and burst-level simulator. The "dense" and "sparse" parameters shown in Table 5.2 on page 139 were used in the experiments undertaken. Each experiment was performed with the **CL**, **CLt**, **BL**, **BL2**, **BLt** and **BL2t** simulators (as described in Section 5.4 on page 133). The cell-level simulation accuracy results (from **CL**) were taken to be the "ideal" results, when comparison with the burst-level simulator results was made.

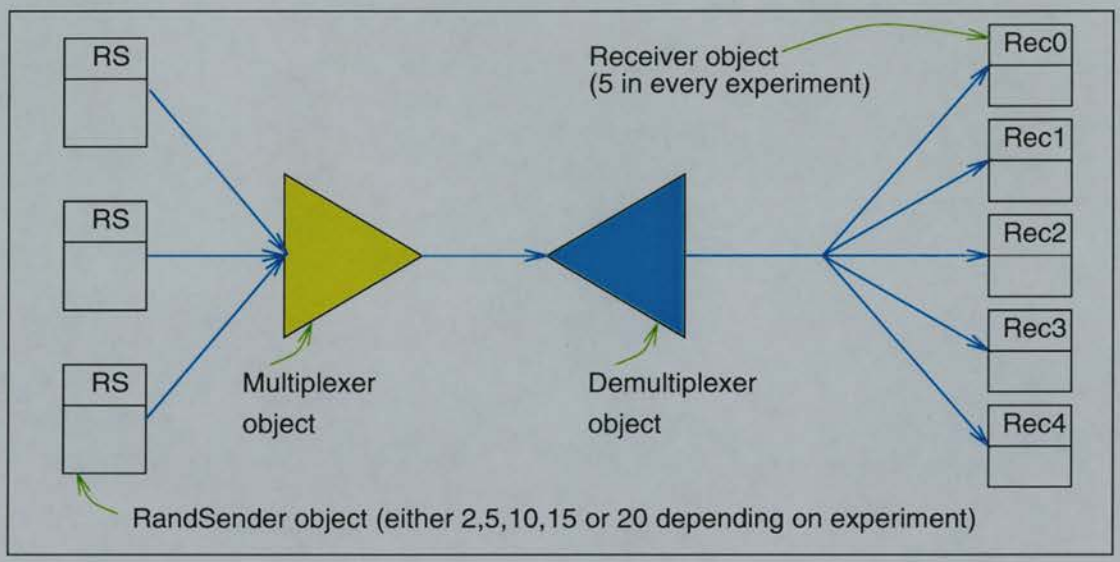


Figure 5.19: The simulator objects used in the demultiplexing experiments

The behaviour of each `RandSender`, `Multiplexer` and `Receiver` object was the same as in the multiplexer and queueing experiments (see Section 5.6.3 on page 136). The minimum output ACTT values for the `Multiplexer`, and each `Demultiplexer` output port, were set such that no queueing was required (ie. the output ACTT value for each object was lower than the minimum possible in each simulation run). This removes the consideration of large memory footprints due to the queueing techniques chosen in each of the cell-level simulators. It also means that the effects on the accuracy and performance of each simulator due to demultiplexing could be seen without the additional complication of cell queueing. Some minor cell queuing *is* required in the cell-level multiplexer as more than one

input cell may arrive at the same simulation time. This small-scale queueing was handled by scheduling cells into the future in the cell-level multiplexers.

As the amount of processing necessary in the burst-level demultiplexer depends on the number of cell streams merged at each output port, it was decided to keep the number of `Receiver` objects constant between experiments simulating different numbers of cell streams. This was to show how the performance and accuracy of the burst demultiplexing scaled with the number of cell streams sent from each output port. Five `Receiver` objects were used, with a random variable used to determine the mapping of each `RandSender` cell stream to each demultiplexer output port. The random variable was drawn from a *Randint* distribution, where each value has equal probability of selection. The mapping was used as a *network destination address* to be set during burst production in each `RandSender` object. The destination address for each `RandSender` was constant throughout an experiment, such that *all* of the bursts produced by one `RandSender` went to the same `Receiver` object. The random mapping depended on the number of cell streams (ie. the number of `RandSender` objects) in each simulation, but experiments with the same number of simulated cell streams had *identical* destination mappings for the `Demultiplexer` outputs. Table 5.16 shows the number of cell streams mapped to each `Receiver` object, depending on the number of cell streams simulated in the experiment. When a `Receiver` object has greater than one cell stream mapped to it, some multiplexing needs to be performed by the `Demultiplexer` at the output port feeding that `Receiver` object. If a `Receiver` only has one cell stream mapped to it, no multiplexing is required at that `Demultiplexer` output port.

	Receiver object				
Streams	Rec0	Rec1	Rec2	Rec3	Rec4
2	1	0	1	0	0
5	1	1	2	1	0
10	1	3	3	2	1
15	1	5	5	2	2
20	3	6	7	2	2

Table 5.16: The number of cell streams received per receiver object in the demultiplexer experiments

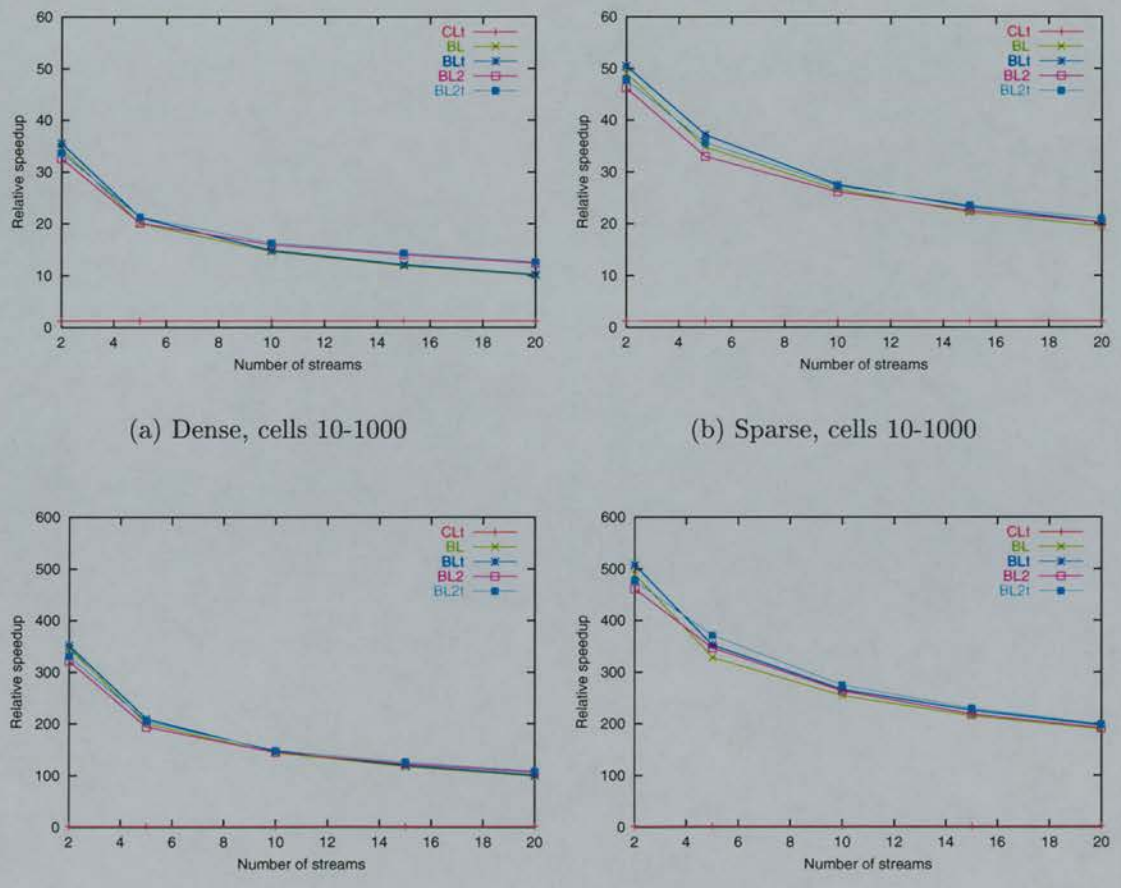
5.8.3 Results

As in the multiplexer and queue experiments, the "dense" and "sparse" experiments were performed in two batches. The first batch was optimised for runtime

performance analysis, whereas the second batch performed runtime analysis on the bursts received by each `Receiver` object such that accuracy analysis could be performed. The runtime results are in Section 5.8.3.1, whereas the accuracy analysis results start from Section 5.8.3.2.

5.8.3.1 Runtime comparison

The relative speedup of each simulator, when compared to the runtimes for the **CL** simulator, are shown in Figure 5.20. The actual data may be seen in Table A.6 in Appendix A. A plot for the post-ordered tree event list cell-level simulator (**CLt**) is included in Figure 5.20, as it was marginally faster than the **CL** simulator in every experiment.



(a) Dense, cells 10-1000

(b) Sparse, cells 10-1000

(c) Dense, cells 100-10000

(d) Sparse, cells 100-10000

Figure 5.20: The relative runtime speedups for each simulator in the demultiplexer experiments

As can be seen from Figure 5.20, the burst-level simulators show a relative speedup when compared with the runtimes of the **CL** simulator. The absolute

difference in runtime speedups between each of the burst-level simulators is very small. The **BL2t** simulator offered marginally the best performance of the four burst-level simulators, especially for the experiments with larger numbers of cell streams. This shows that the simulation time distribution of event insertions was "friendlier" to the post-ordered tree in these experiments than it was in the queueing experiments in Section 5.7.4.1 on page 165. The burst-level speedups are better for the experiments simulating the larger burst sizes, as would be expected. The "sparse" results are also better than the equivalent "dense" results due to the lower number of cell streams per incoming burst to the demultiplexer. The fewer the number of cell streams in the input burst, the less work the demultiplexer has to do to produce the relevant output bursts for the `Receiver` objects.

The amount of work the demultiplexer must perform is related to the number of cell streams merged onto each output port, and the number of output bursts produced per input burst. The demultiplexer must allocate incoming cell streams to each output port and then calculate the burst ACTT for each output burst produced. No burst splitting is required, as the length of each output burst is governed by the size of the input burst to the demultiplexer from which each output burst is produced. As shown in the multiplexing experiments, the performance advantage offered through the use of the improved integer techniques (as used in **BL2** and **BL2t**) is apparent when each burst to be processed contains many cell streams. In the demultiplexing experiments, the number of cell streams at each demultiplexer output port is small compared to the maximum stream count for each input burst. With the relative speedups of the burst-level simulators very close to each other, it is apparent that the job of producing the output bursts is the dominant overhead in the demultiplexer object. If each output port had more cell streams mapped to them, the cost of calculating the burst ACTT for each output burst would have a greater influence. In this case the improved integer technique burst-level simulators would show a greater performance improvement. The performance, in general, will improve if the number of output bursts to produce per input burst is small. This is the effect seen in the relative difference between the "sparse" and "dense" experiments in Figure 5.20.

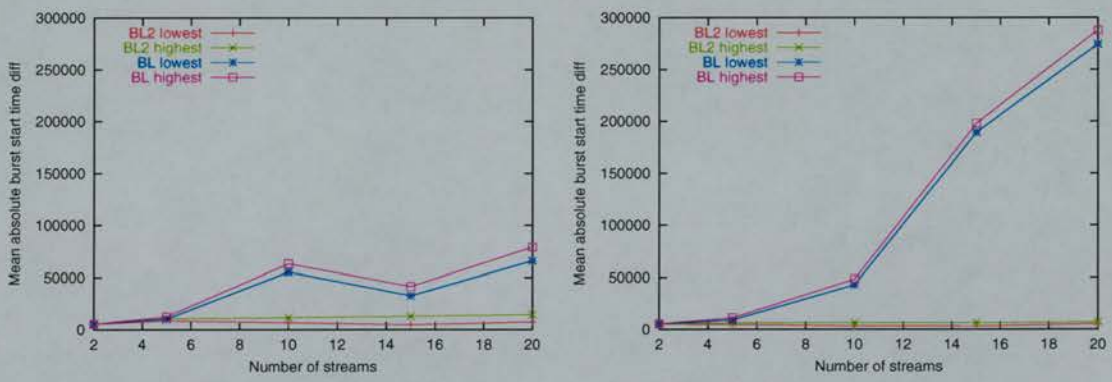
5.8.3.2 Accuracy comparison

The accuracy analysis information required for the results in the following sections was obtained from the second batch of simulator experiments. Each `Receiver` object, in both the cell and burst-level simulations, recorded the burst information as described in Section 5.6.4.2 on page 145. As in the queue and multiplexer

experiments, the results for the post-ordered tree event list simulators were the same as from their indexed event list counterparts, and so are not discussed.

5.8.3.3 Burst start time comparison

The first analysis performed was on the burst start time delay incurred by each burst in the burst-level experiments when compared to the same burst in the equivalent cell-level experiment. The mean absolute start time difference for each burst in each cell stream was computed. As the results were produced for each stream, the minimum and maximum mean absolute burst start time delays were recorded for each experiment set. An experiment set comprised of the results for 2, 5, 10, 15 and 20 streams, with all of the other parameters constant for each experiment. The mean start time delays for the "dense" demultiplexing experiments can be seen in Figure 5.21. The numerical results may be seen in Table A.7 in Appendix A.



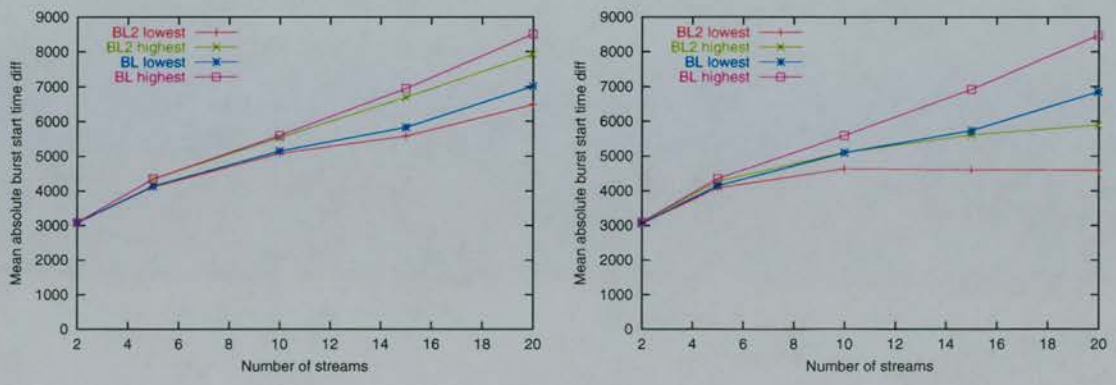
(a) No. cells per burst 10-1000

(b) No. cells per burst 100-10000

Figure 5.21: Lowest and highest mean absolute burst start time differences for the burst-level simulations in the "dense" burst production experiments

The results shown in Figure 5.21 are entirely consistent with those in the multiplexing and queueing experiments. Once again the **BL2** simulator benefits from the ACTT underestimate for each burst produced by the multiplexer. As the bursts produced by each demultiplexer output port are entirely dependent on the arrival of input `START` and `FINISH` messages from the multiplexer, any cumulative delay incurred at the multiplexer output is passed on to the bursts produced by the demultiplexer. However, the demultiplexer may itself incur delay at each output port depending on the ACTT value it calculates for each output burst. The strict time ordering of `START` and `FINISH` messages cannot be compromised. If an output `START` is ready before the scheduled time of the last `FINISH` message

for an output port, the START message will be delayed. The **BL2** simulator is more able to recover from such delays, as it will also underestimate burst ACTT values for demultiplexer output bursts containing more than one stream.



(a) No. cells per burst 10-1000

(b) No. cells per burst 100-10000

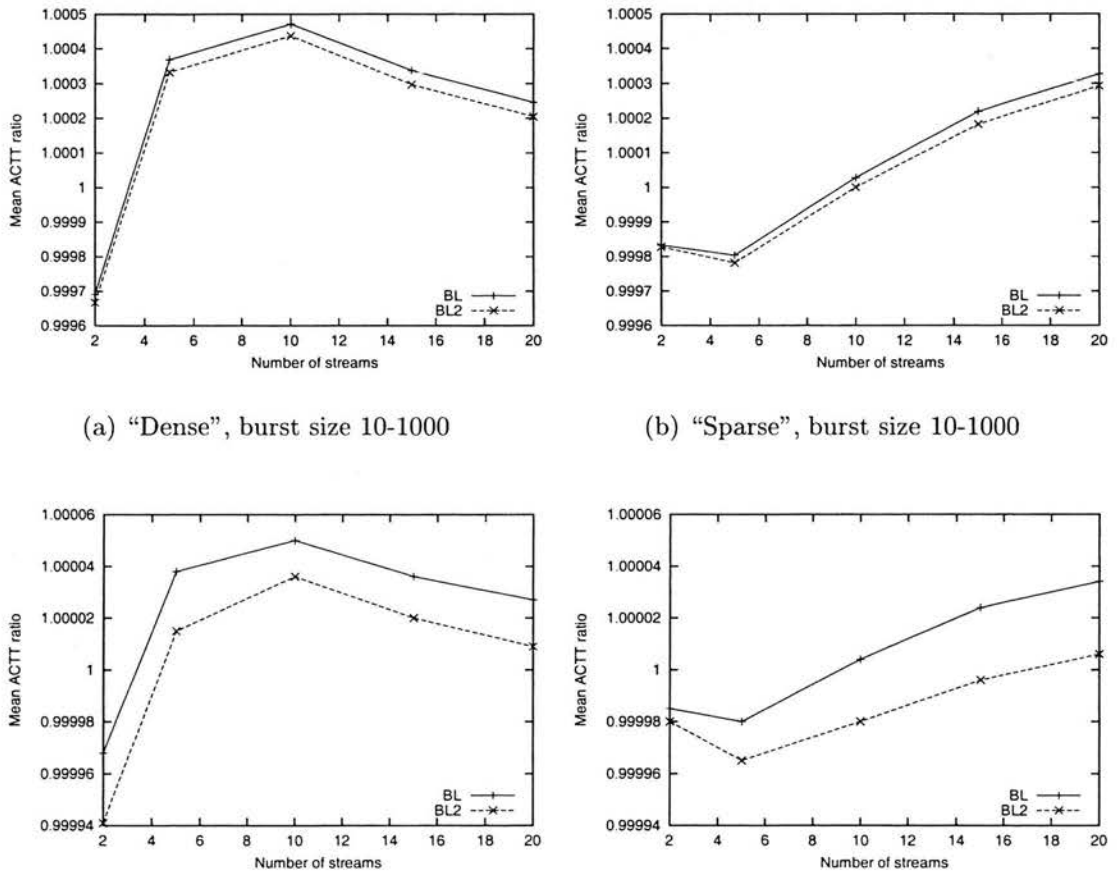
Figure 5.22: Lowest and highest mean absolute burst start time differences for the burst-level experiments in the "sparse" burst production experiments

The results for the "sparse" experiments are shown in Figure 5.22, with the numerical results available in Table A.8 in Appendix A. The difference between the ranges of the **BL** and **BL2** results is small in the "sparse" experiments, although the magnitude of the mean start delay is much smaller than in the "dense" experiments. The bursts produced by the multiplexer in the "sparse" experiments have a lower average number of cell streams than in the "dense" experiments. This enables the multiplexer output to recover from any cumulative delay, and this directly influences the burst timings of the demultiplexer. The difference between the mean start delays in the "sparse" results, where the burst size range is 100-10000 cells, increases with increasing stream count. This is due to the larger burst sizes meaning that the burst duration is a greater underestimate at each demultiplexer output port. This enables the demultiplexer to recover from any delay more easily than in the lower burst size experiments. As the number of streams increases, the number of bursts to be multiplexed at the demultiplexer output ports increases. This increases the scope for underestimation of the burst ACTT at each output.

5.8.3.4 Burst ACTT comparison

The next stage of the analysis was to compare the final burst ACTT values for each experiment in the burst and cell-level simulations. This was achieved by computing an ACTT ratio for each burst received by the `Receiver` objects,

and then by computing an average (with standard deviation) over all of the bursts in each experiment. This was done to give an overview of the effects of demultiplexing for each simulator type. The ACTT ratio computed was the final ACTT value for a burst in a burst-level experiment, divided by the final ACTT value for the same burst in the equivalent cell-level experiment. The average ACTT ratios for each experiment type can be seen in Figure 5.23. The numerical ACTT ratio results (showing the standard deviations) can be seen in Table A.9 in Appendix A.



(a) "Dense", burst size 10-1000

(b) "Sparse", burst size 10-1000

(c) "Dense", burst size 100-10000

(d) "Sparse", burst size 100-10000

Figure 5.23: Mean ACTT ratios for all of the bursts received at the `Receiver` objects in the demultiplexing experiments

The ACTT ratio results for the demultiplexing experiments are consistent with those in the multiplexing and queueing experiments. Once again, the experiments using the larger burst size range showed mean final ACTT ratios closer to 1 than in the shorter burst size experiments. This shows that the bursts describing more cells are a better approximation of the multiplexed streams of cells as modelled

in the cell-level simulations. The longer burst sizes mean that any errors in the ACTT values can be averaged over the burst. When the burst size is small, the effects of having cells added by fractions of cell contributions becomes more apparent, and this can lead to an overestimate of the ACTT for a particular burst. This problem is highlighted by the demultiplexer, as it splits bursts back into their individual cell streams (each of which is also a burst). Any error in the duration of each component stream, when compared to the duration of the original multiplexed burst, will then be apparent.

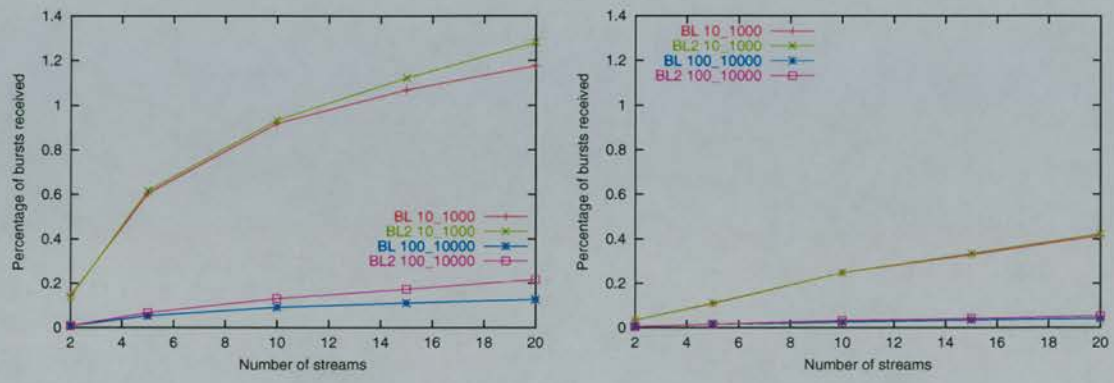
The demultiplexer attempts to make output burst durations match the duration of the input burst which produces them by altering the ACTT for each output burst. If a demultiplexer output burst contains only one stream, the ACTT for the burst is set such that the burst will be *at least* of equal length to the input burst. This calculation can be made as the demultiplexer knows the maximum duration of the input burst from the input `START` message, as well as the maximum cell count for the stream. If the demultiplexer has to increase the ACTT for an output burst fragment, the net effect will be to increase the ACTT for the *entire* burst. Any ACTT overestimate can also lead to scheduling delays at the demultiplexer outputs. This effect is seen as a general ACTT ratio overestimate in the results. This overestimate increases with the number of cell streams in the "sparse" experiments, and decreases (after reaching a maximum) with increasing number of cell streams in the "dense" experiments. This difference is due to the effects of adding cells by fractions averaging over the streams in the "dense" experiments, as the number of streams per burst is larger than in the "sparse" experiments. In the "sparse" experiments, when the number of input cell streams to the multiplexer increases, the sparse nature of burst arrivals keeps the number of streams per burst small. This means that errors in the cell stream durations with respect to the burst duration, will not benefit from the overall averaging seen in the "dense" experiments.

The final average ACTT ratios are smaller in the **BL2** results than in the equivalent **BL** results. This is due to the ACTT underestimate, made both at the multiplexer and at each of the demultiplexer output ports, when more than one cell stream is sent per burst.

5.8.3.5 Zero and overlong burst comparisons

The final analysis performed for the queueing experiment results was to find the number of zero cell length and overlong bursts produced. A zero cell length burst is one in which the `FINISH` message marks the burst as describing zero

cells in total. An overlong burst is one in which the FINISH message arrives at a simulation time after the burst duration (ie. the product of the total number of cells and the burst ACTT) plus the arrival time of the START message for the burst.



(a) "Dense" burst production

(b) "Sparse" burst production

Figure 5.24: The percentage of zero length output bursts produced by the demultiplexer in the "dense" and "sparse" experiments

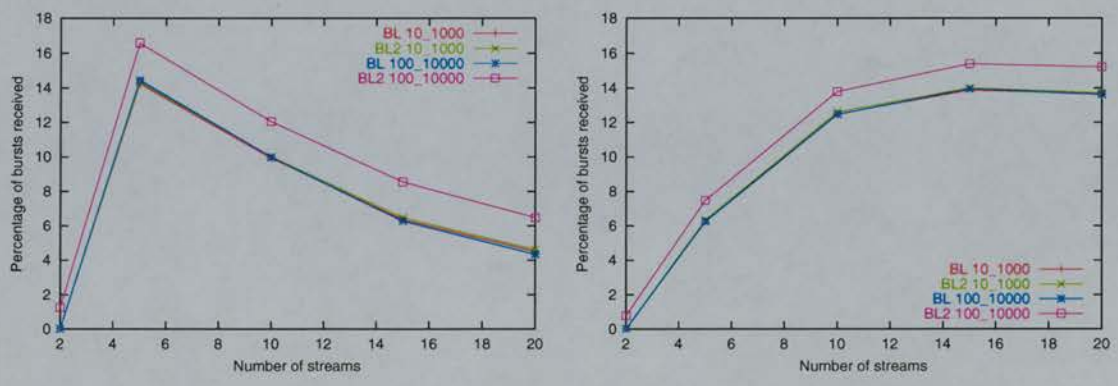
Figure 5.24 shows the total percentage of zero length bursts received at all of the Receiver objects in each of the experiments. As can be seen from the results, the proportion of zero cell length bursts increases with increasing stream count, especially in the "dense" experiments where the burst size range was 10-1000 cells. As the burst-level demultiplexer contains no buffering or a parameterised delay at each output port, zero cell length bursts are difficult to remove. If some delay has been accumulated at the output port due to ACTT overestimation of a previous burst, a zero cell length burst can be cancelled. However, if the START message has already been scheduled, there is no choice but to send a FINISH message, even if it describes a zero cell length burst.

The proportion of zero cell length bursts is highest in the "dense" experiments for the burst length range of 10-1000 cells due to the short output bursts produced by the multiplexer in this case. As the source bursts describe a small number of cells, and the multiplexer is "busy" due to the dense arrival of input bursts, the net effect is to produce output bursts describing a small total number of cells. When the demultiplexer processes these, the input START message may contain non-zero cell counts for each component stream in the burst. If a stream has a zero cell count in the START message, the demultiplexer can prevent the inclusion of this stream in the destination output port burst. When the input FINISH message arrives, any streams which have a zero cell length will require a FINISH

to be sent at their demultiplexer output port. As the number of input streams to the multiplexer increases, the number of zero cell length component streams in each output burst increases, especially for input bursts describing few cells. This effect is seen as an increasing zero cell length proportion with an increasing number of cell streams for each experiment in Figure 5.24. When the number of cells per burst is increased, a drop is seen in the proportion of zero cell length bursts received at the `Receiver` objects. This is consistent with the output bursts produced by the multiplexer containing fewer zero cell length component streams.

The proportion of zero cell length bursts is smaller in the "sparse" experiments when compared to the "dense" results. This is due to the smaller number of cell streams per output burst produced by the multiplexer. As would be expected, the proportion of zero cell length bursts reduces with increasing initial burst size (ie. the results when the burst cell size is 100-10000 cells).

Figure 5.25 shows the percentage of overlong bursts received by all of the `Receiver` objects in each of the demultiplexing experiments. As can be seen from the results, the burst-level demultiplexers can produce a high proportion of overlong bursts from their output ports. The results for each of the burst-level simulators are very close, apart from **BL2** which produces a higher proportion of overlong bursts for the larger burst size experiments.



(a) "Dense" burst production

(b) "Sparse" burst production

Figure 5.25: The percentage of overlong (wrt. expected burst duration) output bursts produced by the demultiplexer in the "dense" and "sparse" experiments

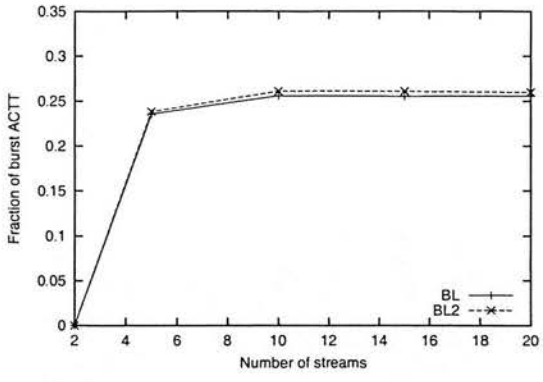
The "dense" results (Figure 5.25(a)) show a reduction (after reaching a maximum) in the proportion of overlong bursts with increasing cell stream count. This is due to each input burst to the demultiplexer containing more cell streams, which means each output port will in turn send more cell streams. The greater the number of merged cell streams, the closer the burst ACTT is to the ideal for

the multiplexed cell stream. In this case, the value computed by the demultiplexer for each output burst is closer to the ideal for each burst, so that the duration of each output burst more closely matches that of the source input burst. When this happens the number of overlong bursts produced will decrease. The results are slightly worse for the **BL2** simulator as the calculation of burst ACTT values leads to an underestimate of the burst ACTT. In this case, when the input `FINISH` message arrives at the demultiplexer, the durations of the output port bursts will be slightly "too short" due to the ACTT underestimate. Hence, when the `FINISH` message is generated at the demultiplexer output port, the result is an overlong burst.

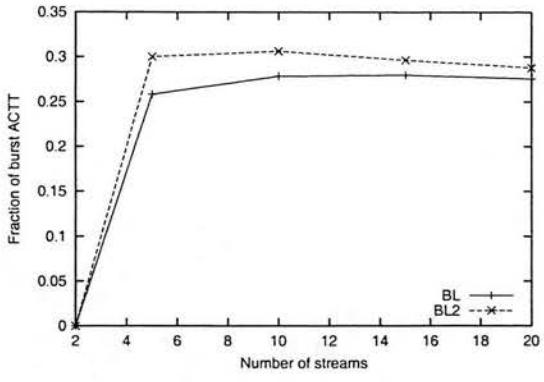
The "sparse" results (Figure 5.25(b)) show an increase in the number of overlong bursts with increasing cell stream count. This is due to the smaller average number of cell streams contained in each multiplexer output burst, when compared to the "dense" experiments. For a smaller number of cell streams in each input burst, the demultiplexer sends fewer cell streams per burst produced at each output port. The net effect is that the duration of each burst at the output port mismatches the duration of the source input burst. Once again, the **BL2** results are slightly worse due to the consistent burst ACTT underestimation produced by the revised integer techniques.

The degree of zero cell length and overlong burst production by the demultiplexer can be offset by adding queues to each output port. This has the advantage of simulating output buffers, as well as reducing the number of zero cell length and overlong bursts produced. To assess the likely impact of adding queues, the burst-level simulators were annotated to record the amount by which each overlong burst was longer than the maximum duration described in the `FINISH` message. For each overlong burst produced, the absolute difference between the calculated `FINISH` message time and the actual scheduled time was recorded. This time difference was then divided by the burst ACTT to give the delay as a fraction of a cell. The cell fractions were then averaged for all of the overlong bursts in each experiment. Figure 5.26 shows the average cell fraction by which bursts were overlong in each of the demultiplexer experiments. The numerical versions of the results may be seen in Table A.10 in Appendix A.

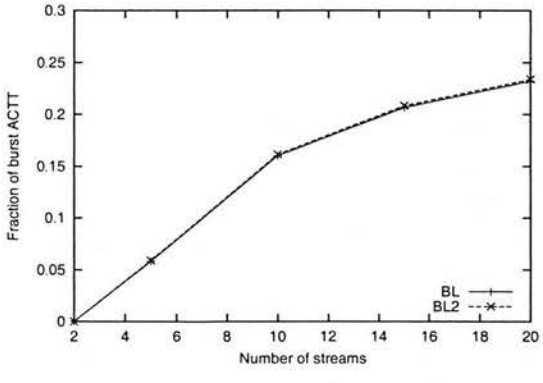
As can be seen from Figure 5.26, each overlong burst is "too long" by a modest fraction of a cell. In the "dense" experiments the average overlong cell fraction is almost a constant with increasing cell stream count. The "sparse" results show an increase in the average overlong cell fraction due to the lower number of cell streams per burst. With the degree by which each burst was "overlong" being



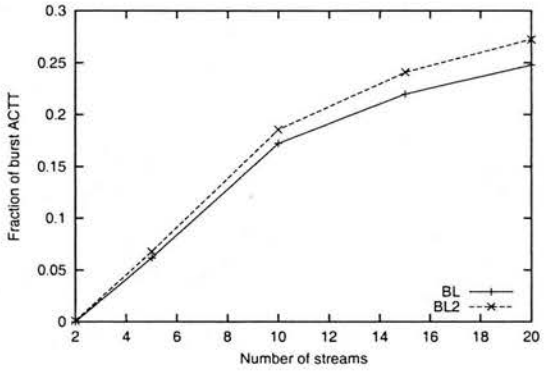
(a) "Dense", burst size 10-1000



(b) "Dense", burst size 100-10000



(c) "Sparse", burst size 10-1000



(d) "Sparse", burst size 100-10000

Figure 5.26: Mean average `FINISH` time delay (expressed as a fraction of the burst ACTT) for each overlong burst received at the `Receiver` objects.

a small fraction of a cell, the likelihood of output port `Queue` objects reducing the proportion of overlong bursts is good, especially with a parameterised delay added to each queue output.

5.9 Summary of chapter

The main aim of this chapter was to assess the relative performance and accuracy implications of using the core burst-level simulation objects introduced in Chapter 3. Another aim was to evaluate the impact of using the improved integer techniques described in Chapter 4 in a real simulation environment. These aims were achieved by performing three sets of experiments focusing on the `Multiplexer`, `Queue` and `Demultiplexer` objects in turn. An efficient cell-level simulator, also developed in the work and designed to use the same simulation framework as the

burst-level simulators, provided the "benchmark" runtime and accuracy results with which each burst-level simulator could be compared.

The two areas of investigation were the runtime performance and the accuracy of each of the techniques. The choice of the burst size ranges modelled in each burst-level simulation has the greatest bearing on the performance of the techniques. The trade-off made in the decision on an appropriate burst size is speed versus accuracy. If a modeller decides that a low resolution model is acceptable in terms of the results it will produce, then large burst sizes can be simulated. If, however, the model under consideration requires a finer resolution of detail in the results produced, much smaller burst sizes are required. The larger the simulated burst size range, the greater the number of cells represented by each burst and the less work the burst-level simulation objects have to perform.

The findings presented allow for some conclusions and guidelines to be given on the use of the burst-level simulation objects for performing modelling work.

5.9.1 Runtime performance

5.9.1.1 Burst size range

In terms of the potential simulation runtime speedup offered by the burst-level simulators, the most critical factor is the burst size, in cells, of each burst simulated. The burst-level simulation objects analysed in this chapter all benefit from larger burst sizes when compared with the runtimes of the same experiments performed in a cell-level simulator. The greater the burst size, the larger the possible runtime speedup available.

5.9.1.2 Time "density" of burst arrivals

While a cell-level simulator will have a total event count which increases with an increasing number of cell transmissions to be simulated, the burst-level simulators do not have such a clear cut dependency. Instead, the degree of burst fragmentation caused by multiplexing, queueing and demultiplexing has an influence on the total event count and hence the runtime of a simulation. This fragmentation is in turn influenced by the time density of burst arrivals at each simulation object. This was demonstrated in the analysis by the runtime speedup differences between the "dense" and "sparse" burst arrival experiments. When an object has to deal with frequent burst arrivals, the work required of the object is increased. In the case of the multiplexer, the fragmentation of each burst will also increase. The greater the number of bursts produced by a simulation object, the greater the amount of work other simulation objects will have to subsequently perform

to process the bursts. The more work the burst-level simulator has to do, the smaller the runtime speedup when compared with the same cell-level simulation experiment.

5.9.1.3 Choice of global event list data structure

The simulation framework itself also has a bearing on the potential speedup available. As was dramatically shown in the queueing experiments, the performance of the post-ordered tree event list was very much worse than that of the indexed doubly-linked list. In the other experiments, the post-ordered tree was shown to be slightly faster in nearly every case. This inconsistency makes it hard to choose one technique over the other, as predicting where the performance will suffer is difficult. The performance of the indexed event list was shown to be influenced by the maximum number of events assigned to each index. Some crude estimate of the potential maximum number of pending events needs to be made before setting this value in a simulation. This estimate must be based on the number, and type, of objects in each simulation with factors such as the degree of multiplexing possible having a major influence.

In the cell-level simulators, the post-ordered tree event list offered marginally better performance in almost every experiment when compared to the indexed event list version. This shows that the simulation time locality of event insertion into the tree was consistently good for the post-ordered tree. In the burst-level simulations, such temporal locality of simulation time insertions cannot be guaranteed, as events mark burst `START` and `FINISH` times spanning possibly many cells. With this in mind, use of the indexed event list structure is recommended, as the maximum number of events per index can be fine tuned when feedback from previous experiments is available. The performance of the indexed event list was always close to that of the post-ordered tree, so any fine-tuning will always have a slight effect on performance if the first estimate of the maximum index event count is good.

5.9.1.4 Bypassing the global event list

Another performance improvement technique, that of bypassing the global event list, was shown also to have a small, but perceptible, influence on the experiment runtimes. Such a technique is possible in a bespoke simulation environment, as any behavioural insight can be directly applied to the objects being modelled. This is in contrast to a well-tuned, but general purpose, simulation environment where such tweaks may be difficult or impossible. As a performance benefit was

seen, any such benefit will be magnified over larger simulation runs, leading to lower runtimes.

5.9.1.5 Using the revised integer techniques

Another factor in the speed up obtainable was the use of the improved integer techniques in the burst-level simulation objects. The revised techniques were shown to give some performance improvement, but the degree of this improvement was small apart from when many cell streams were simulated. This is due to amount of integer division performed in each object being dependent on the component stream count in each burst. The objects which benefited the most from the improved integer techniques were the `Multiplexer` and `Queue` as they perform the most integer arithmetic on each burst, depending on the number of streams. The performance improvement was small, as other overheads inherent in the simulator dominate performance. The effort required for functions such as the event list management and the processing of events may allow a clever compiler, or even a dynamic instruction scheduler in the microprocessor, to "amortize" the cost of expensive integer divides over other code execution. Such complications reduce the performance benefit seen through the use of the improved techniques. Even though the scale of the performance improvement was small, any technique which improves runtime performance is still valuable especially in large simulations.

5.9.2 Result accuracy

5.9.2.1 Final burst ACTT ratios

The results for the final average ACTT ratios for each burst were generally consistent across the experiments. Once again the burst size (in cells) has a major influence on the range of final burst ACTT ratios. The larger the bursts simulated, the closer the average ACTT ratio is to the ideal of 1.0, as well as having a lower standard deviation. This is due to each multiplexed burst being a better representation of the multiplexed stream of cells it represents as the burst size increases. Any error, as introduced by burst splitting, is averaged over the larger number of cells in each cell stream. The use of the **BL2** simulator leads to slightly smaller final average ACTT ratios than the **BL** simulator, due to the burst ACTT underestimation encountered.

5.9.2.2 Minimising burst START time delays

Of the accuracy analysis results presented, the most striking difference between the burst-level techniques used was for the mean burst start time difference seen in each experiment. As described in Chapter 3, the calculation of the burst ACTT is designed to be an overestimate to ensure that bursts *at least* describe the transmission duration of the cells they represent. As seen in the results, this overestimate can lead to large delays in the start times for bursts when compared with the equivalent cell-level simulation experiment results. This delay is due to cumulative delays at the multiplexer output port as burst messages "queue up" behind previous messages delayed due to ACTT overestimation. This effect is most noticeable in the "dense" experiments due to the frequent burst arrivals at the multiplexer. When the multiplexer has "sparse" burst arrivals, the delay at the output can be recovered in any inactive output transmission time. Any delays introduced by the demultiplexer are faithfully mirrored by the queue and demultiplexer objects, so removing the delay at source is advantageous.

When the results from the **BL2** simulator were analysed it was apparent that a "side-effect" of the techniques used helped reduce the average burst start time delay. As this simulator uses the improved integer techniques, it was found to *underestimate* the burst ACTT values calculated in the multiplexer. Although the degree of the underestimate was small, the effect was enough to prevent cumulative message delays building at the multiplexer output port. The knock-on effect was to see reduced average start time delays in both the queueing and demultiplexing experiments also. The `Queue` object can also help recover any incurred message delay through the queueing operation, but the demultiplexer is completely at the mercy of the multiplexer. With this effect in mind, and the other benefits of using the improved integer techniques for ACTT calculation (see Chapter 4), the **BL2** simulator is recommended for any experiments where dense burst arrival is possible.

5.9.2.3 Accuracy of burst-level queueing

The queueing results showed the relative performance of the burst-level queues when cell loss was encountered. On the scale of the total number of cells lost in each experiment, and the degree of loss encountered by each cell stream, the results from **BL** and **BL2** were very close to those obtained in the cell-level simulations. Although the cell losses simulated were artificially high (compared to anticipated losses in cell-based networks), this was done to show the limitations of the burst-level queueing techniques implemented. What was found was that

when the final burst sizes were compared, the variation in the results from the burst-level simulators increased with an increasing number of cell streams per burst. The variation also increased as the burst size decreased. This suggests that queue-derived cell loss is better for larger burst sizes, especially when analysis at the burst scale is required. This is once again due to larger burst sizes being a closer representation of the multiplexed stream of cells they represent.

5.9.2.4 Minimising zero cell length and "overlong" bursts

A side effect of the burst-level techniques is the production of zero cell length and "overlong" bursts. Zero cell length bursts represent wasted effort in a simulation, whereas overlong bursts can affect the accuracy of burst start times and final burst ACTT values. As was seen in the multiplexer and especially the demultiplexer experiments, the percentage of zero cell length and overlong bursts produced could be significant. However, the queue experiments showed that the actions of queueing bursts, as well as utilising a parameterised transmission delay, could dramatically cut the number of zero cell length and overlong bursts produced.

A delay need not necessarily be due to a queue object, as transmission distances can also be modelled as delays in the scheduling of burst messages. The magnitude of any delay determines how many zero cell bursts can be removed and the reduction, or elimination, of overlong bursts. Considering the potentially large percentages of zero cell length and overlong bursts produced by the demultiplexer, it would be advisable to add a `Queue` object, or a parameterised delay, to each output port. This would be of extra benefit when using the **BL2** simulator as it produced more overlong bursts than the **BL** version. If each output port queue did not use flow control, then the event list could be bypassed to help reduce the computational cost.

Chapter 6

Experimental performance

6.1 Introduction

The aim of this chapter is to show how the simulation objects described in Chapter 3 perform when combined to produce non-trivial simulation models. The experiments presented are used to assess the effects of the approximations made in the techniques when many inter-connected objects are involved in a model. The performance and accuracy issues for each object in isolation were shown in Chapter 5, so the experiments in this chapter show how these issues influence the results of more realistic simulations. The results can then be used to guide further work on the technique and give extra guidance for modellers wishing to use integer-time burst-level simulation.

Three basic experiments are presented to show the simulation objects operating in different simulation scenarios:

1. The first experiment is an example of a non-trivial structure of interconnected simulation objects.

2. The second experiment examines the performance of the simulator under a high loading scenario.

3. The third experiment is an example of the simulator modelling very high bandwidth communication channels.

The simulation model used in the first experiment is a small cell-based network where end-to-end performance of a simple communicating pair of network nodes is monitored. Other nodes provide interfering cross traffic which influences the delay experienced and any cell loss. Simulation studies are presented for both the cell and burst-level simulators, so that comparisons can be made between the results.

The second example models a large number of telephone calls sharing a single communications link. The experiments are performed with the **BL** and **BL2** burst-level simulators to show how the runtime performance of each varies with the total number of simultaneous telephone calls modelled in each simulation. The focus is on the burst-level techniques as a large number of concurrent cell stream bursts have to be dealt with.

The third example models a local area network of workstations connected by very high speed links. This set of experiments shows how the simulation techniques and objects developed in this thesis can be used to model a network which is *not* based on the transmission of cells. This is a change of emphasis for the simulator, as the techniques were developed with cell-based communication networks in mind. However, if the burst can be used to represent a quantity in some other network type, the simulation techniques and objects described in this thesis can be used to model that network type. The minimum transmission unit of interest in the model described in the third example in this chapter is the *byte* rather than a cell. For this model, the burst is defined as a packet of bytes all transmitted at the same data rate. This model demonstrates the ability of the integer burst-level techniques to simulate very high speed data transfers. The experiments for this model are performed using the **BL** simulator only, as little or no multiplexing occurs.

Finally, a summary of the chapter is presented.

All of the experiments conducted in this chapter were performed on the same workstation as used for the investigations in Chapter 5. The configuration of the workstation is described in Section 5.5 on page 134.

6.2 An example cell-based network

To show how the simulation objects introduced in Chapter 3 can be used to model telecommunications networks, a simple network model was produced. The model contains four switches and ten communicating nodes. A simple *constant bit rate* (CBR) server sends streams of cells to receiving partners, after receiving communication requests from the partners (a similar video-on-demand simulation model is presented by Srivastava[97]). *Variable bit rate* (VBR) cross traffic also flows through the switches, and this determines any delay or cell loss incurred by the CBR traffic. The network model is described in detail in the next section.

6.2.1 The network model

Figure 6.1 shows the object structure of the network model used in each experiment. The **CBRsource** object is capable of sending up to ten concurrent cell streams into the network. The **Mux** object merges the cell streams from the **CBRsource** to produce an input to the cell switch ***Sw0***. Each **CBRsink** initiates a cell stream transmission from the **CBRsource** object by sending a request message to it over the network (the message being a burst of five cells). When the **CBRsource** receives a request from a **CBRsink**, it firstly allocates a free send channel, and then sends a stream of bursts of cells to the **CBRsink**. Each **CBRsink** waits for a random time (as set by a parameterised random distribution) before sending the next request message to the **CBRsource** after a current transmission has ended.

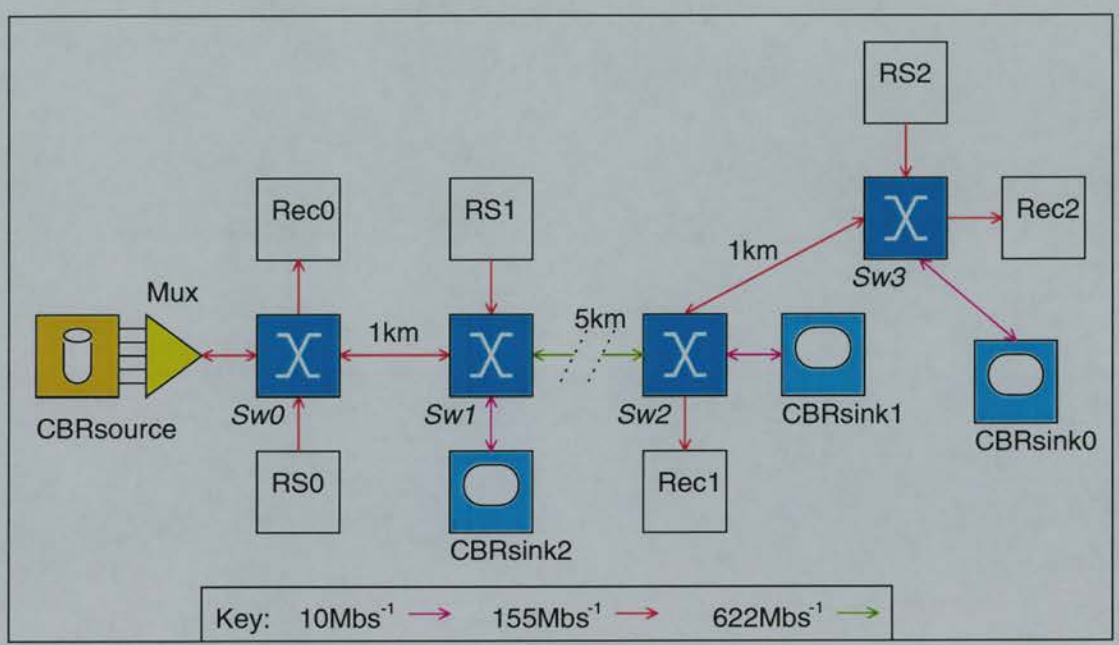


Figure 6.1: The simple network model used in the experiments

Variable bit rate cross traffic is generated by the **RS** objects and received by the **Rec** objects. Each **RS** object is similar to the **RandSender** object described in Section 5.6.3 on page 136, in that it sends variable length (in terms of the total number of cells) bursts, each with an ACTT value chosen from a parameterised random distribution, with a parameterised delay between adjacent bursts. Each **Rec** object is a simple receiver which just accepts streams of cells from the network. The destination **Rec** object for each **RS** object was fixed for all of the experiments, with the mappings as follows:

- **RS0 → Rec2**

- **RS1 → Rec1**
- **RS2 → Rec0**

The fastest link to be simulated in the model is 622.08Mbs^{-1} (ie. ITU-T STM-4) between switches ***Sw1*** and ***Sw2***. To keep parameters consistent between the cell and burst-level simulators, the time to transmit one cell at 622.08Mbs^{-1} was set to be 1000 units of the integer simulation clock (of type `unsigned long long int`). This is equivalent to setting one unit of the clock as the time to transmit one cell at this rate, but using an ACTT multiplier of 1000 (as described in Chapter 4). Table 6.1 shows the conversions necessary for simulating the line speeds shown in the model in Figure 6.1. For calculating the transmission times between each switch, it was assumed that fibre optic cabling was used and that the transmission speed in each cable was approximately 2×10^5kms^{-1}. The distance between each node and its switch was assumed to be negligible as regards any implicit delay. The model does not explicitly quantify any interface delays (ie. caused by the transfer of information between the electrical and optical domains), but each output port queue adds a small delay to model the transfer of data through the queue.

Transmission speeds	
Value	Integer clock units
622.08Mbs^{-1}	1000
155.52Mbs^{-1}	4000
10Mbs^{-1}	62208
1.5Mbs^{-1}	414722
Transmission delays	
Fibre length	Integer clock units
1km	7336
5km	36679

Table 6.1: The conversion of "real world" metrics to integer clock units for each simulation

The traffic generators were deliberately kept very simple, as the development of detailed communication traffic models is outside the scope of this work. Any traffic models from the relevant literature, which can suitably be represented by bursts, could be easily coded and added as simulation objects. The simple traffic models chosen in this model are used to illustrate the performance and accuracy issues of using the burst-level simulation environment described in this thesis.

6.2.2 Simulating the model

6.2.2.1 The burst-level simulators

The simple network model was implemented in both the **BL** and **BL2** simulators as described in Chapter 5 (see Section 5.4.2). An indexed event list was used in preference to the post-ordered tree list, due to the slight performance differences for the burst-level simulators seen in Chapter 5.

Each cell switch was a compound switch comprising multiplexers, demultiplexers and queue objects (without flow control). Such a compound switch is described in Section 3.7 on page 76. Output buffering was enabled for each switch, with the event list bypass mechanism (see Section 5.7.4.1) enabled for burst transfers between each output multiplexer and output port queue object. Each queue was configured to introduce a delay of one output ACTT to bursts traversing the queue. If a line transmission delay was required for an output port (to model the transmission delay along fibre optic cables) this was added at the appropriate switch output port queue object.

Although each switch is a compound entity, a simple `switch_new` class is used to instantiate and configure each switch. Figure 6.2 shows the code necessary to configure switch ***Sw0*** in one of the experiments performed. The high-level `switch_new` class takes care of instantiating and interconnecting the correct number of multiplexer, demultiplexer and queue objects (if buffering is configured) necessary to implement a $N \times N$ switch. The complete simulation model code for one of the experiments performed is shown in Section B.1 in Appendix B.

6.2.2.2 The cell-level simulators

The simple network model was implemented for both the **CL** and **CLt** simulators (see Section 5.4.1), as the post-ordered tree event list was shown to offer benefits to the performance of the cell-level simulator.

Figure 6.3 shows the internal structure of the cell-level switch used in each experiment. A cell-level switch is the functional equivalent of a cell-level demultiplexer (as described in Section 5.8.1 on page 181). Cells are routed to an appropriate output port buffer by comparing the *destination address* of each cell with the *port mapping table* which maps valid addresses to switch output ports. Each internal output queue in the cell-level switch object used physical storage for buffering cells, rather than the forward scheduling technique. As in the burst-level simulator, a queue throughput delay of the value of one output ACTT was added to cells traversing the switch output buffer. Line transmission delays were also appropriately added by each switch output port affected.

```
//Instantiate 4x4 switch with output buffering and event list bypass
sw[0] = new switch_new("Sw0", sim_ev_list, 4, TRUE, TRUE);
...
//Set output port parameters for Switch 0 (port, ACTT, delay, capacity)
sw[0]->SetOPQueue(0, 4000, 4000, 1000);
sw[0]->SetOPQueue(1, 4000, 4000, 1000);
sw[0]->SetOPQueue(2, 4000, 7337+4000, 1000);
sw[0]->SetOPQueue(3, 4000, 4000, 1000);

//Set input and output object links for each switch port
mux[0]->SetLink(sw[0]->GetInPort(0));       //Input from Mux
sw[0]->SetOutPort(0, cbrsrc[0]);            //Output to CBRsource

sw[0]->SetOutPort(1, rec[0]);               //Output to Rec0

sw[0]->SetOutPort(2, sw[1]->GetInPort(0)); //Link to next switch

rs[0]->SetLink(sw[0]->GetInPort(3));        //Input from RS0

//Build port mapping table (dest address, sw output port)
sw[0]->AddressSize(20);
sw[0]->SetAddress(4, 1);   //Address of Rec0
sw[0]->SetAddress(5, 2);   //Address of Rec1
sw[0]->SetAddress(6, 2);   //Address of Rec2
sw[0]->SetAddress(10, 0);  //Address of CBRsource
sw[0]->SetAddress(11, 2);  //Address of CBRsink0
sw[0]->SetAddress(12, 2);  //Address of CBRsink1
sw[0]->SetAddress(13, 2);  //Address of CBRsink2
```

Figure 6.2: Instantiating a switch in the burst-level simulator

6.2.3 The experiments

6.2.3.1 Experimental parameters

Depending on the experiment, each burst stream channel generated by the **CBR-source** was either 1.5Mbs^{-1} or 10Mbs^{-1}. At 1.5Mbs^{-1}, each transmission approximates 10 seconds of a broadcast quality video signal. At 10Mbs^{-1}, each transmission could be considered as an *FTP* transfer of the equivalent video clip over standard *Ethernet*. Each channel was configured to send 10 time sequential bursts, each carrying 3500 cells, when a request from a **CBRsink** was received. A *negative exponential* distribution, with a mean of 1.0, was used by each **CBRsink** to determine the number of seconds ("real time" seconds) between the end of the previous transmission and the scheduling of the next request burst message to the **CBRsource**. This generated a chain of frequent requests and transmissions for each **CBRsink** across the network.

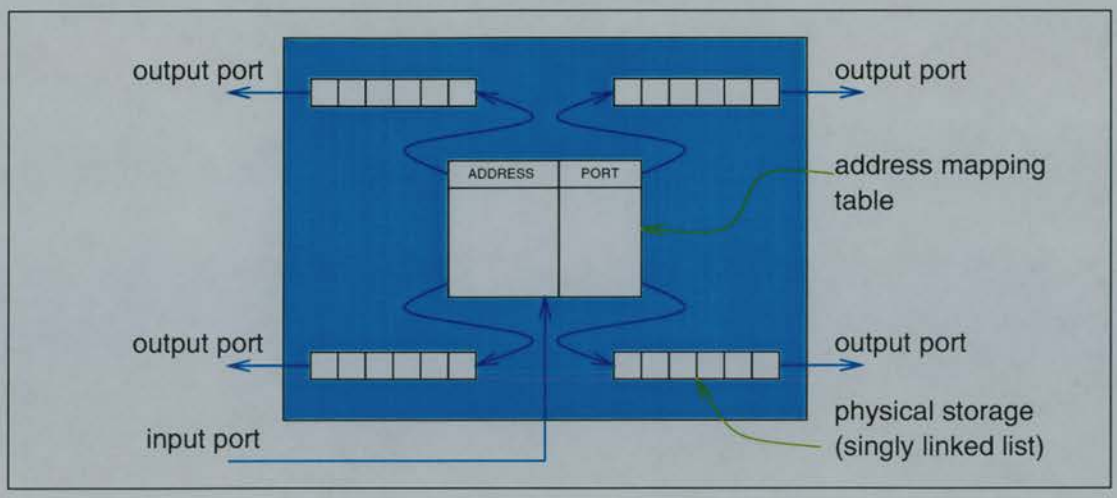


Figure 6.3: A 4x4 cell-level switch

Each **RS** and **CBRsink** object was given a "final simulation time" parameter which controlled the length of each simulation run. If the current simulation time was greater than the final simulation time, each **RS** object would cease generating bursts and each **CBRsink** object would cease generating request bursts. The final simulation time was set to be 2×10^{12}, which is approximately 23 minutes of "real" network time (one second is approximately 1467179201 units of the integer clock).

The burst generation parameters for each VBR source (ie. each **RS** object) are given in Table 6.2. Like the **RandSender** objects in Chapter 5, each VBR source used a "gap multiplier" to produce a delay between adjacent bursts. The parameters chosen for the ACTT value of each burst meant that the streams of bursts produced by each **RS** object ranged between 10Mbs^{-1} and 155Mbs^{-1} in terms of their data transmission bit rates.

Setting	Value
Cells	*Randint*(1000,100000)
ACTT	*Randint*(4000,62210)
"Gap" multiplier	*Uniform*(0.0,2.0)

Table 6.2: The parameters for each VBR source in the experiments

As previously mentioned, each output port queue (or buffer in the cell-level experiments) introduced a queue latency to each cell stream equal to the value of one minimum ACTT value for the queue. The capacity of each output queue (or buffer) in each experiment was set to 1000 cells. Even if each switch port output buffer was fully occupied with cells, the memory requirements of the cell-level simulator for this model would be modest with the capacities set to 1000 cells.

All of the experiments described in the following sections were performed on the same workstation. Care was taken to ensure that the workstation was unloaded by other user processes. To get averaged values for each of the parameters of interest, five repetitions of each experiment for each simulator type were undertaken. The seed for the random number generator was altered for each repetition of the same experiment. Where appropriate, results are reported with a 95% confidence interval calculated from the average results produced in each experimental run. Mitrani[83] and MacDougall[77] detail the technique used to find the confidence intervals. In the result tables in the following sections, each 95% confidence interval is shown as "$A \pm H$", where A is the average and H is the confidence interval *half-width* for the average result A presented. This means that, as calculated using the five average results from each experiment, there is a 95% confidence that the true average lies between $A - H$ and $A + H$.

6.2.3.2 Experiment 1: one CBRsink (1.5Mbs^{-1}) and no cross traffic

The first experiment involved setting the channel bandwidth of the **CBRsource** to 1.5Mbs^{-1}, and activating **CBRsink0** only (ie. **CBRsink1** and **CBRsink2** were deactivated). No cross traffic from the VBR sources was included.

The results recorded for each simulator type were as follows:

- The runtime (in seconds) of each experiment.
- The final average ACTT value over all of the bursts received at the **CBRsink0** object (the ACTT multiplier is removed by dividing the result by 1000).
- The average delay between the end of the transmission of the request burst and the arrival of the first cell (or `START` message) from the resulting transmission from the **CBRsource**. The ACTT multiplier is removed from this result by dividing the value by 1000. The delay is a measure of the network latency.

Table 6.3 gives the results (averaged over the five repeated runs) of each parameter recorded.

As would be expected, the burst-level simulators offer a vast speedup over the cell-level simulators in this example, while maintaining the accuracy of the results. As no multiplexing, demultiplexing or queueing is required for the stream of bursts in the burst-level simulations, no source of inaccuracy is introduced. The **BL2** simulator is slightly slower than the **BL** simulator, as no streams are multiplexed, thus the need to calculate left shifted reciprocals becomes an overhead.

Simulator / Parameter	CL	CLt	BL	BL2
Run time	53.07	46.79	0.33	0.35
ACTT	186.92	186.92	186.92	186.92
Delay	414.72	414.72	414.72	414.72
Speedup (over **CL**)	-	1.13	160.81	151.63

Table 6.3: The averaged results for the first experiment

6.2.3.3 Experiment 2: three CBRsink objects (1.5Mbs^{-1}) and no cross traffic

The next experiment was similar to Experiment 1, but each of the three **CBRsink** objects were enabled. Once again, no VBR cross traffic was simulated. The same parameters were recorded as in Experiment 1, but for each of the three **CBRsink** objects. Table 6.4 gives the results (averaged over the five repeated runs with the calculated 95% confidence intervals shown) for each parameter recorded.

Simulator / Parameter	CL	CLt	BL	BL2
Runtime	163.07	132.31	1.69	1.77
CBRsink0 ACTT	414.720	414.720	414.722 ±0.0005	414.721 ±0.0007
CBRsink0 Delay	186.97 ±0.04	186.97 ±0.04	910.93 ±176.69	573.62 ±84.46
CBRsink1 ACTT	414.720	414.720	414.722 ± 0.0005	414.721 ± 0
CBRsink1 Delay	164.27 ±0.02	164.27 ±0.02	895.09 ±177.55	544.70 ±75.85
CBRsink2 ACTT	414.720	414.720	414.722 ±0.0005	414.721 ±0.0005
CBRsink2 Delay	88.92 ±0.02	88.92 ±0.02	547.02 ±110.74	230.40 ±13.75
Speedup (over **CL**)	-	1.23	96.55	92.19

Table 6.4: The averaged results for Experiment 2

As can be seen from Table 6.4, the burst-level simulators offer a speedup relative to the cell-level simulators. The final average ACTT values for each of the **CBRsink** objects in the burst-level simulators are slight overestimates of the ideal values, as reported in the cell level results. Since no cell loss is incurred, or delay introduced by interfering cross traffic, one would expect the final ACTT values to be very close to their original values. Simulator **BL2** reported slightly lower average ACTT values than **BL** as it *underestimates* the burst ACTT values (as described for the experiments in Chapter 5).

Where the burst-level results are not so good is for the average delay between the finish of a request message departure, and the arrival of the first START message (from the **CBRsource**) for that transmission. However, in terms of "real" time a delay of 573.62 (for **CBRsink0** in **BL2**) is approximately 4×10^{-4} seconds, which is very small. Considering that the bandwidths of each of the links modelled is ample for the traffic produced in this simulation model, one would expect that there would be no delay over and above the network latency (also very small in terms of real time). This is the case seen for the average delays reported in the cell-level results. **CBRsink2** is closest to the **CBRsource** object and so has the smallest delay. Larger delays are experienced by the **CBRsink0** and **CBRsink1** objects as they are 4 and 3 switches distant from the **CBRsource** respectively. The burst-level experiment average delays reflect the relative differences in the delays experienced by each CBRsink, but the magnitudes of the actual delays are greater than for the cell level results.

The reason for the larger delays in the burst-level simulators is related to the multiplexing operation at the **Mux** object. Burst START and FINISH messages can only be aligned to cell transmission boundaries as each burst must describe an integral number of cells. The net effect of this requirement may be to introduce delays for new cell streams when their START messages arrive at the multiplexer input ports. If the multiplexer is currently sending an output burst when the new START message arrives, a FINISH message must be generated and sent at the output port at a simulation time reflecting the length of the output burst (ie. product of the total number of cells carried and the burst ACTT). If the time calculated for the transmission of the FINISH message is greater than the arrival time of the new START message, then the output START message including the details of the new burst will be delayed. Such a delay can be seen in the burst-level results in Table 6.4. The size of a delay incurred through this effect is related to the absolute size of the burst ACTT (ie. a large burst ACTT and a delay of just one cell will delay the subsequent burst by that large ACTT value) and the number of streams currently being multiplexed (ie. the greater the number of streams, the smaller the burst ACTT). Experiment 3 in Section 6.2.3.4 explores the differences made when the ACTT is reduced (ie. the bandwidth increased) for each channel from the **CBRsource**, when the other parameters are identical to Experiment 2.

Although both burst-level simulators show delay overestimates, the averages for the **BL2** simulator are lower than those for the **BL** simulator. The major reason for this is in the underestimation of the burst ACTT values when the re-

vised integer techniques are used. The underestimation was shown in Chapter 5 to reduce delays for each of the main simulation objects. Delays can accumulate at the output port of an object when the `FINISH` message for an output burst is scheduled at a greater simulation time than the arrival time of an input `START` message. This is an important factor in the multiplexer object, as the multiplexer finishes transmission of an input burst when it receives the `FINISH` message for that burst. If the output burst ACTT is an overestimate, building a burst containing an appropriate number of cells such that all cells from the input burst will have been sent, may cause a delay at the output port. If input burst arrivals are "dense" in time at the multiplexer, the delay accumulates as the multiplexer cannot recover the time synchronisation of output message creation with input message arrivals. When the burst ACTT is an underestimate, as in the **BL2** simulator, large accumulated delays at the output port are less likely. This results in the lower magnitude delays, compared with the **BL** results, seen in the results. However, the "cell alignment" delay at the **Mux** is still evident, and this is reflected by the absolute difference between the **BL2** and **CL** average delays being less than one original ACTT value for each stream (ie. 414.72 excluding the ACTT multiplier).

6.2.3.4 Experiment 3: three CBRsink objects (10Mbs^{-1}) and no cross traffic

Experiment 3 was identical to Experiment 2 in the previous section, apart from setting the bandwidth for each channel in the **CBRsource** to 10Mbs^{-1}. The same parameters were recorded as for Experiment 2. Table 6.5 gives the results (averaged over the five repeated runs with the calculated 95% confidence intervals shown) for each parameter recorded.

As can be seen from Table 6.5, the burst-level simulators offer a large speedup over the cell level variants. Once again, both burst-level simulators overestimate the average ACTT and the average delay for each **CBRsink**. However, the magnitude of each overestimated delay is very much reduced when compared to the results seen for Experiment 2 in Table 6.4. The smaller burst ACTT values result in smaller output delays for each object as cell alignment delays are reduced. The smaller ACTT values mean that the burst ACTT values calculated in the **BL2** simulator are less likely to be underestimates, hence the similarity of the delay results between the **BL** and **BL2** simulators.

Simulator / Parameter	CL	CLt	BL	BL2
Runtime	897.28	783.31	6.33	6.70
CBRsink0 ACTT	62.21	62.21	62.23 ±0.05	62.23 ±0.05
CBRsink0 Delay	187.09 ±0.04	187.09 ±0.04	217.26 ±0.92	206.26 ±0.56
CBRsink1 ACTT	62.21	62.21	62.21 ±0	62.21 ±0
CBRsink1 Delay	164.40 ±0.02	164.40 ±0.02	194.55 ±1.59	183.24 ±0.53
CBRsink2 ACTT	62.21	62.21	62.21 ±5e-05	62.21 ±4e-05
CBRsink2 Delay	89.06 ±0.02	89.06 ±0.02	111.29 ±0.89	103.77 ±0.71
Speedup (over **CL**)	-	1.14	141.75	133.92

Table 6.5: The averaged results for Experiment 3

6.2.3.5 Experiment 4: three CBRsink objects (1.5Mbs^{-1}) and VBR cross traffic

The next experiment included interfering VBR cross traffic generated by the **RS** objects. Each of the three **CBRsink** objects was enabled, and the **CBRsource** channel bandwidth was set to 1.5Mbs^{-1}. The parameters used for configuring each VBR source are shown in Table 6.2 on page 206. No cell loss was encountered (or indeed was expected), so the parameters recorded were the same as for Experiment 2. Table 6.6 shows the results (averaged over the 5 repeated runs and with the calculated 95% confidence intervals shown) for each parameter recorded.

The average runtime for the **CL** simulator was *slower* than real time, as the simulated time is 1380 seconds (23 minutes). The other simulators offer better than real time speedups, with the burst-level simulators being very much faster than real time.

The effect of the interfering cross traffic can be seen in increased **CBRsink** delay averages for each simulator, as would be expected. The increase in the average delay for each cell-level simulator is slight (cf. the results in Table 6.4), but very much increased for each burst-level simulator. The combined effects of cell alignment at each multiplexer, as well as burst ACTT overestimation and delays due to finishing input bursts on receipt of their `FINISH` message are responsible. The average delays in the **BL2** simulator are lower than in the **BL** simulator, but still significantly larger than in the cell-level results. This is a side-effect of the number of simulation objects each burst must traverse. For a burst travelling

Simulator / Parameter	CL	CLt	BL	BL2
Runtime	1401.84	952.19	3.12	3.25
CBRsink0 ACTT	414.720	414.720	414.740 ±0.002	414.733 ±0.001
CBRsink0 Delay	187.22 ±0.06	187.22 ±0.06	4755.34 ±2168.51	1314.25 ±223.59
CBRsink1 ACTT	414.720	414.720	414.739 ±0.001	414.732 ±0.002
CBRsink1 Delay	164.55 ±0.08	164.55 ±0.08	4615.63 ±2174.72	1163.94 ±235.80
CBRsink2 ACTT	414.720	414.720	414.729 ±0.0009	414.727 ±0.0009
CBRsink2 Delay	89.19 ±0.08	89.19 ±0.08	732.46 ±325.05	357.99 ±21.36
Speedup (over **CL**)	-	1.47	449.31	431.33

Table 6.6: The averaged results for Experiment 4

from the **CBRsource** to the **CBRsink0** object, for example, one multiplexer and four switches must be traversed. To traverse a switch, an burst must pass through a demultiplexer (ie. the input port), a multiplexer and then finally a queue (ie. the output port). If other cells streams are added or removed at each stage, the scope for adding inaccuracy and delay increases. The interfering cross traffic provides such a source of inaccuracy.

A particular influence on the increase in the average delays seen in each burst-level simulator is the relative ACTT size difference between the CBR bursts and the VBR cross traffic bursts. Each CBR burst has an original ACTT of 414.72, whereas each VBR burst can vary between 4.0 and 62.1 (excluding the ACTT multiplier). This means that cell alignment delays have a greater effect. If a "slow" CBR burst delays a "fast" VBR burst at an output port, a large number of VBR cells can accumulate. This is not a problem in the queue object, but the multiplexer does not perform queueing internally. Instead, the multiplexer records the arrival time of the VBR burst, so that when it must generate output bursts including the VBR burst, the cell counts are correct. Output burst `FINISH` messages are generated when either a burst `START` or `FINISH` arrives at the multiplexer. When this happens, the cell count for the VBR burst is calculated from its arrival time at the multiplexer. Producing the appropriate `FINISH` message to include this number of cells (delayed by the previous cell alignment or accumulated output port delay) will incur a further delay at the output port. If the next `START` message includes a new CBR stream, it will suffer this delay, hence the large delays seen in the burst-level simulators. Underestimation of the output

burst ACTT helps to keep the accumulated delay in check, hence the smaller average delays seen in the **BL2** simulator.

6.2.3.6 Experiment 5: three CBRsink objects ($10Mbs^{-1}$) and VBR cross traffic

Experiment 3 showed that the average delays seen by each **CBRsink** in the burst-level simulators were reduced when the bandwidth of each **CBRsource** channel was increased. With this in mind, Experiment 5 is a repeat of Experiment 4, but with each **CBRsource** channel set to $10Mbs^{-1}$. This modification, when combined with the VBR cross traffic, means that there is the risk of cell loss and queueing at the output port of ***Sw0*** which links to ***Sw1***. The peak bandwidth from the **CBRsource** ($30Mbs^{-1}$) coupled with the peak bandwidth of **RS0** ($155Mbs^{-1}$), is larger than the $155Mbs^{-1}$ link connecting ***Sw0*** and ***Sw1***. With this in mind, *cell loss ratio* (CLR) information was recorded for switch ***Sw0*** and each of the **CBRsink** objects. The cell loss ratio is defined as the ratio of lost cells to the original number of cells before the loss.

Table 6.7 gives the results for each of the parameters recorded for Experiment 5. Averages are over the five runs of each experiment, and the calculated 95% confidence interval for each average is also shown (when non-zero).

For Experiment 5, both cell-level simulators were slower than real time (ie. taking longer than 23 minutes to execute), whereas both burst-level simulators were substantially faster than real time.

As was expected, cell loss occurred at the output port of switch ***Sw0*** which was connected to switch ***Sw1***. Each of the simulators was in good agreement as to the overall magnitude of the cell loss ratio reported by switch ***Sw0***. This is in agreement with the results presented in Section 5.7.4.3 on page 171, where the burst-level queue object was shown to give good agreement on the cell loss encountered at a queue at the macroscopic scale. The burst-level cell losses are not in such good agreement for each of the **CBRsink** objects. However, the averages considered with the confidence intervals, overlap for both the cell and burst level results. This is in agreement with the results in Section 5.7.4.3 which showed that the burst-level queue object could show wide variation in the distribution of cell loss between concurrent cell streams traversing the queue. It should be noted that the half-widths of the confidence intervals calculated for each CLR are large relative to the average value. This suggests that either longer simulation runs, or more experiments, should be performed to give better confidence in the CLR results presented.

Simulator / Parameter	CL	CLt	BL	BL2
Runtime	2376.78	1684.45	9.51	10.01
Sw0 CLR($\times 10^{-5}$)	19.597 ±14.827	19.597 ±14.827	19.520 ±14.475	19.594 ±14.707
CBRsink0 ACTT	62.219 ±0.006	62.219 ±0.006	62.250 ±0.057	62.250 ±0.058
CBRsink0 Delay	189.26 ±3.42	189.26 ±3.42	317.36 ±9.55	258.84 ±3.26
CBRsink0 CLR($\times 10^{-5}$)	13.004 ±9.752	13.004 ±9.752	4.119 ±3.103	4.099 ±3.148
CBRsink1 ACTT	62.218 ±0.005	62.218 ±0.005	62.232 ±0.0456	62.214 ±0.002
CBRsink1 Delay	165.82 ±2.97	165.82 ±2.97	281.95 ±10.45	224.64 ±4.76
CBRsink1 CLR($\times 10^{-5}$)	11.014 ±8.005	11.014 ±8.005	4.203 ±3.121	4.222 ±3.175
CBRsink2 ACTT	62.219 ±0.008	62.219 ±0.008	62.233 ±0.046	62.233 ±0.045
CBRsink2 Delay	89.32 ±0.038	89.32 ±0.038	123.00 ±1.550	119.92 ±1.170
CBRsink2 CLR($\times 10^{-5}$)	12.343 ±12.639	12.343 ±12.639	5.069 ±3.801	5.098 ±3.882
Speedup (over **CL**)	-	1.41	249.92	237.44

Table 6.7: The averaged results for Experiment 5

Other factors will also influence the microscopic cell loss allocation at the queue. One consideration is that the cell-level simulators used are also approximations of the behaviour of real hardware. Subtle behavioural quirks, which for example may be side-effects of the discrete event techniques used, may bias the allocation of loss to cells arriving at the cell-level queue. Of particular importance to the burst-level queues is the mix of cell streams which arrive, and the burst splitting technique used to apportion cell loss over the streams. The average delays experienced by the CBR cell streams (due to the multiplexing technique and the VBR cross traffic) will influence the composition of the bursts to be queued. Experiment 6 (see Section 6.2.3.7) explores this possibility by making the **CBRsource** autonomously send cell streams to "dumb" versions of the **CBRsink** objects.

As was the case when comparing the results of Experiment 3 with Experiment 2, the increase in the channel bandwidth reduces the average delays seen in Table 6.7. Once again, the **BL2** simulator shows smaller average delays than the **BL** simulator (due to burst ACTT underestimation), but the results of both are closer to the average delays shown in the cell level results.

The effect of the interfering VBR cross traffic is to slightly increase the delays seen in the **CL** simulators. Delays due to the cross traffic also lead to a slight increase in the average ACTT value reported in each simulator. The increase is smaller in the cell-level results than in the burst-level results, but the burst level results show the comparative increase (cf. Experiment 3 when there is no interfering VBR cross traffic).

6.2.3.7 Experiment 6: three CBRsink objects ($10Mbs^{-1}$), VBR cross traffic and autonomous CBRsource

All of the previous experiments initiated burst transmissions from the **CBRsource** by sending request messages (bursts of five cells) from each **CBRsink** object over the network to the **CBRsource**. Experiment 6 differed in that the **CBRsource** autonomously sent burst streams to each of the **CBRsink** objects. The idea was to remove the delay associated with the request messages sent by each **CBRsink** over the network. Each **CBRsink** in these experiments was effectively a "dumb" receiver, giving no feedback to the **CBRsource**. Delays between subsequent transmissions to each **CBRsink** were governed by the same negative exponential random distribution, as used in the **CBRsink** objects, with a mean of 1.0 second. This meant that each **CBRsink** received frequent transmissions, with small time gaps between each transmission.

To help improve the confidence of each average CLR result reported, the final simulation time was increased to 4×10^{12} units of the global simulation clock, which represents approximately 46 minutes of real time. Five repetitions of each experiment were performed, with the random number generator seed altered for each one. The parameters recorded for each experiment were the same as for Experiment 5. Table 6.8 shows the results (averaged over the 5 runs and with the computed 95% confidence intervals shown) for each simulator type for Experiment 6.

Once again, the burst-level simulators offered a much better than real time performance, whereas the cell-level simulators performed worse than real time. It should be noted, however, that each **CL** experiment performed processed around 1.5 billion events, at approximately 300000 events per second.

As in Experiment 5, each simulator was in good agreement as to the scale of cell loss experienced at the output port connecting switch ***Sw0*** to switch ***Sw1***. The relative size of the confidence half-width also showed improvement, as a direct consequence of increasing the length of each simulation run. The CLR averages reported for each **CBRsink** showed that on average, the burst-level simulators

Simulator / Parameter	CL	CLt	BL	BL2
Runtime	4951.72	4584.93	18.22	19.10
Sw0 CLR($\times 10^{-5}$)	15.265 ±9.707	15.265 ±9.707	15.284 ±9.714	15.233 ±9.688
CBRsink0 ACTT	62.217 ±0.005	62.217 ±0.005	62.215 ±0.002	62.215 ±0.002
CBRsink0 CLR($\times 10^{-5}$)	10.058 ±6.683	10.058 ±6.683	3.409 ±2.672	3.395 ±2.665
CBRsink1 ACTT	62.217 ±0.005	62.217 ±0.005	62.215 ±0.001	62.215 ±0.001
CBRsink1 CLR($\times 10^{-5}$)	9.213 ±6.758	9.213 ±6.758	3.598 ±2.032	3.584 ±2.023
CBRsink2 ACTT	62.217 ±0.005	62.217 ±0.005	62.214 ±0.002	62.214 ±0.002
CBRsink2 CLR($\times 10^{-5}$)	9.775 ±7.815	9.775 ±7.815	3.891 ±2.496	3.875 ±2.481
Speedup (over **CL**)	-	1.08	271.78	259.25

Table 6.8: The averaged results for Experiment 6

underestimated the cell loss at each **CBRsink** object, relative to the average losses reported in the cell-level simulators. The confidence half-widths for each average are still large relative to the size of each average, suggesting that even longer simulation runs are necessary to achieve good confidence in the CLR results when the CLR is small. This is an accepted problem with cell-level simulation (as mentioned in Chapter 2), and is one of the reasons for research into faster simulation techniques for cell-based networks.

Although the cell losses reported in each experiment are small, the burst-level techniques appear to slightly bias the loss toward the cell streams with the smallest ACTT values. The ACTT values for the VBR traffic can range between 4000 and 62210, whereas each CBR cell stream had an ACTT of 62208 (when the ACTT multiplier of 1000 is included). Bursts containing a mixture of VBR and CBR streams will contain many more VBR cells if the VBR stream ACTT is small, and this may bias the allocation of lost cells to the VBR stream. Effects such as cell alignment delays at the multiplexer, and the associated delays due to having to send the correct number of cells from each input stream, will slightly alter the microscopic make-up of each burst entering the queue. However, the *total* number of cells to enter the queue will be the same in the burst and cell-level simulators, hence the good agreement as regards the macroscopic cell loss at the queue. When the cell losses are small, even a small change in the microscopic composition of the bursts being queued will affect the cell loss allocation among

the streams. This is the effect seen in the CLR results in both Experiment 5 and Experiment 6.

As can be seen from the results, the interfering VBR cross traffic causes the average ACTT for each **CBRsink** to increase. This increase is seen across all of the simulators, but with the average slightly higher in the burst-level simulators (although the burst-level averages lie within the confidence intervals for each cell level result).

6.2.4 Summary of findings

The experiments presented in this section were designed to show how the burst-level simulation objects described in Chapter 3 could be combined to perform a simple network simulation. Good performance speedups were possible with the burst-level simulators, over equivalent cell-level simulations, for each experiment undertaken. A summary of the large runtime speedups obtained through the use of the burst-level simulators (when compared to the runtimes of the cell-level **CL** simulator) is shown in Table 6.9. The speedups for each of the **BL2** experiments were slightly lower than those for the **BL** experiments. This was because the number of streams multiplexed in each simulation run was small, and the revised integer techniques have been shown to deliver a performance benefit only when there are a large number of streams being multiplexed. However, each burst-level simulator performed much faster than "real time" (in terms of the time period simulated) whereas the cell-level simulators were slower than real time for the larger simulation experiments.

Experiment	BL speedup	BL2 speedup
1	160.81	151.63
2	96.55	92.19
3	141.75	133.93
4	449.31	431.33
5	249.92	237.44
6	271.78	259.25

Table 6.9: The relative runtime speedups of the burst-level simulators when compared to the **CL** simulator runtimes for each of the cell-based network experiments.

The experiments highlighted the problem of delays incurred by bursts when they are multiplexed. The problem is especially acute when the difference in magnitude of the ACTT values for the bursts to be multiplexed is large. The experiments demonstrated that delays incurred in each multiplexer (whether independent or in a compound switch object) can have a bearing on the results.

The use of the **BL2** simulator using the revised integer techniques was shown to help reduce the absolute magnitude of the delays. It was also shown that having well-matched ACTT values of the cell streams to multiplex also has a favourable effect on reducing any delay. Modification of the multiplexing technique may also help to reduce implicit delays, and this is suggested as an area of future investigation in Chapter 7.

The simple network model used in the experiments had a large switch to transmission distance ratio. If larger physical distances were modelled between each switch, the network latency would increase. If, for example, the distance between switches ***Sw1*** and ***Sw2*** was 200km rather than 5km, the transmission delay would be 1467160 simulation clock units (including the ACTT multiplier of 1000). Such a delay would help to reduce the effect of multiplexer induced delays for the bursts transmitted in the burst-level simulators. The smaller the transmission delays, the more dominant the delays introduced by the multiplexer operation will be.

As was the case in the queueing experiments in Chapter 5, the burst-level simulators were shown to be good at predicting the scale of cell loss at the macroscopic scale, in substantially shorter runtimes than the equivalent cell-level experiments. The fine detail of the cell loss at the burst scale was not so accurate, but the results and the reported confidence intervals overlapped for the cell and burst level results. As was demonstrated in the experiments performed, achieving a satisfactory confidence in the CLR results requires many repetitions of long simulation runs. This is a daunting prospect if using cell-level simulation, but is not such a problem when considering the relative speedups achievable by the burst-level simulators. Examining the delay issues in the multiplexing technique may well alter the microscopic structure of the bursts produced. Such a change may also influence the microscopic allocation of cell loss to multiplexed bursts. This is an area for future investigation.

6.3 Mass multiplexing example

The aim of this example experiment is to show how the runtime performance of each burst-level simulator responds to heavy loading of concurrent cell stream bursts. The system modelled is that of a large number of simultaneous telephone calls sharing a single communications link. The **BL** and **BL2** simulators are used in the experiments to show the differences made by using the revised integer techniques presented in Chapter 4.

6.3.1 The model

Figure 6.4 shows the simple model used in the experiments performed in this section. Twenty **CBRsource** objects are connected to a single multiplexer object, which sends the resultant multiplexed bursts to a demultiplexer object via a queue. The link between the multiplexer output port and the queue is set to bypass the global event list to reduce the number of events to process. Each **CBRsource** can autonomously send up to one hundred concurrent burst streams (ie. each a separate telephone call) to the multiplexer object. The number of **CBRsource** objects active in a simulation is parameterisable, so that between 0 and a maximum of 2000 concurrent burst streams can be sent to the multiplexer.

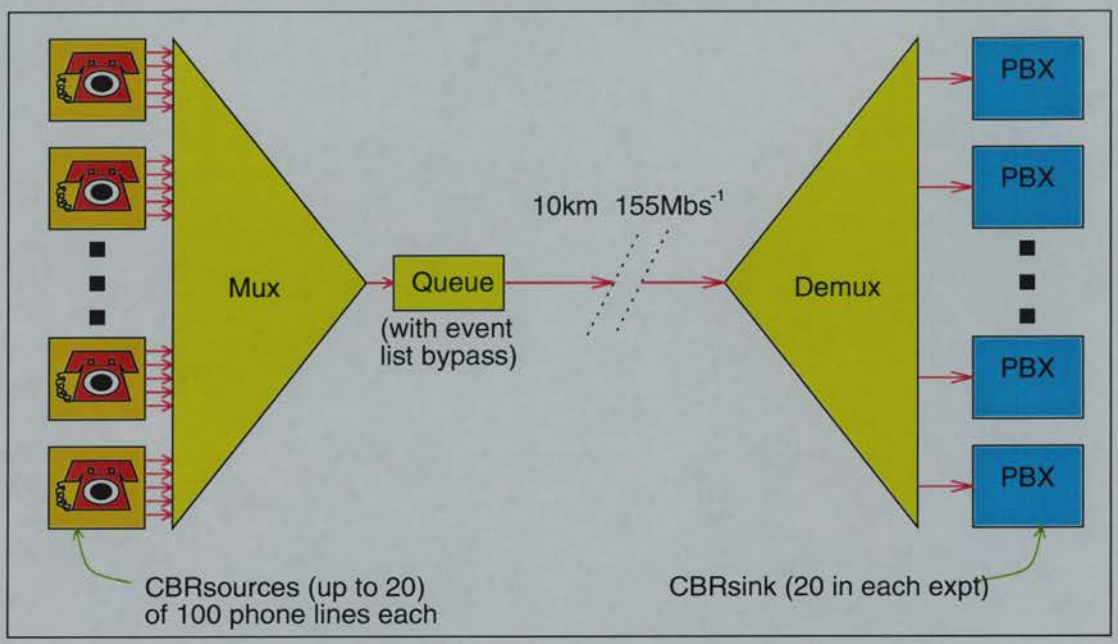


Figure 6.4: The simple telephone network used in the experiments

In each experiment, the demultiplexer object is connected to twenty **CBRsink** objects which passively accept bursts from the demultiplexer output ports. No feedback is sent from the **CBRsink** objects to the **CBRsource** objects, making the data flow unidirectional. Each telephone channel in an active **CBRsource** randomly selects a destination **CBRsink** from the 20 available (ie. using a $Randint(0, 19)$ distribution). As no further demultiplexing of the bursts received by each **CBRsink** occurs, each **CBRsink** object may be thought of as a *Private Branch eXchange* (PBX) which routes multiplexed phone calls on to some other system. Each **CBRsource** may also be considered as a PBX, but in this example each is a *multichannel* source, hence the different symbol used in Figure 6.4.

The communications link modelled has a bandwidth of 155.52Mbs^{-1}, transmitting over a distance of 10km (assuming that the link is a fibre optic cable). Each telephone channel is assumed to transmit at a constant bit rate of 64Kbs^{-1}. The conversions necessary for simulating these values are shown in Table 6.10. Each telephone channel is modelled as an *on-off* source, where the timespan of each phase is drawn from a negative exponential random distribution. The length (in minutes) of each phone call is drawn from a negative exponential distribution with mean value 20.0, whereas the length of each pause between successive calls is drawn from a negative exponential distribution with mean value 30.0. At 64Kbs^{-1}, one minute of a telephone call is approximately 10000 cells (assuming a cell is of length 53 octets, with a data payload of 48 octets, and ignoring any other protocol overhead in each cell). Each telephone call is modelled as a stream of successive bursts, each of length 10000 cells, with the number of bursts equal to the length of the call in minutes.

Transmission speeds	
Value	Integer clock units
155.52Mbs^{-1}	1000
64Kbs^{-1}	2430000
Transmission delays	
Fibre length	Integer clock units
10km	18399

Table 6.10: The conversion of "real world" metrics to integer clock units for each simulation

6.3.2 Experiments and results

The experiments performed involved simulating 8 hours of "real time" for varying numbers of active **CBRsource** objects. The simulation length was controlled by passing each **CBRsource** object a simulation "finish time", after which no new telephone calls could be initiated. Each experiment was performed with both the **BL** and **BL2** simulators, to highlight the performance differences between the integer techniques used. All of the experiments were performed on the same workstation with care taken to ensure that it was unloaded by other user processes. Figure 6.5 shows the runtimes, in seconds, for each of the burst-level simulators when the number of active **CBRsource** objects was varied between 2 and 20 (ie. between 200 (maximum) and 2000 (maximum) concurrent calls). An estimate of the runtime necessary for each experiment if the **CL** simulator were used for this model is also presented. The estimated **CL** runtime can be

calculated by summing the number of cell transmissions reported between each object in each burst-level experiment and then using this number as the minimum number of events which would need to be processed. An estimated **CL** runtime (in seconds) is produced by dividing the minimum number of events by 300000 (the number of events per second processed by simulator **CL** in Section 6.2.3.7).

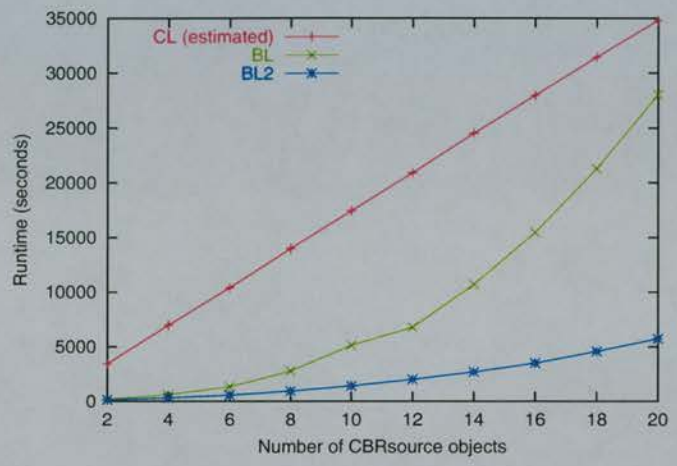


Figure 6.5: The experimental runtimes for the BL and BL2 simulators. An estimate of the runtime required for each experiment using the CL simulator is also shown.

As can be seen from Figure 6.5, the revised integer techniques used in simulator **BL2** offer a large performance advantage when the number of concurrent cell streams is large. This is as would be expected, as the overhead of calculating the burst ACTT for each multiplexer output burst increases with the number of component streams. As was shown in Section 4.5.3.2 on page 117, the original integer burst ACTT calculation method requires an increasing number of integer divide operations as the number of streams increases. The revised integer technique also requires an increasing number of instructions with increasing number of cell streams, but the instructions have lower latencies and throughputs than integer division. Hence, the performance difference between the techniques increases with an increasing number of cell streams. The results from the experiments performed demonstrate this effect.

To examine the effect of mass multiplexing on the ACTT values for each component stream, each **CBRsink** calculated an average ACTT for each stream in each burst received in the entire simulation. With twenty average ACTT values produced in each experiment, an average final ACTT value could be computed for an experiment, along with a 95% confidence interval for each average. Table 6.11 presents the average ACTT values (with calculated 95% confidence intervals) for **BL** and **BL2** for each experiment. Each ACTT value was considered without the

ACTT multiplier of 1000 when calculating the average.

Simulator	BL	BL2
No of active **CBRsource** obj	Average ACTT	Average ACTT
2	2429.79 ± 0.26	2429.91 ± 0.26
4	2429.99 ± 0.16	2430.24 ± 0.16
6	2429.86 ± 0.13	2430.27 ± 0.14
8	2429.44 ± 0.13	2429.93 ± 0.14
10	2428.51 ± 0.14	2428.94 ± 0.14
12	2427.98 ± 0.09	2428.70 ± 0.09
14	2426.64 ± 0.09	2427.40 ± 0.09
16	2425.76 ± 0.09	2426.51 ± 0.09
18	2424.16 ± 0.09	2424.96 ± 0.09
20	2422.84 ± 0.09	2423.71 ± 0.09

Table 6.11: The final average ACTT values (with calculated 95% confidence interval) for each telephone channel in each experiment performed

As can be seen from Table 6.11, the average burst ACTT decreases from the ideal value of 2430 as the number of concurrent telephone channels increases. This is due to the resolution of the burst ACTT for a multiplexed burst worsening with an increasing number of component cell streams. ACTT multipliers (as described in Section 4.2.5.1 on page 87) are used to increase the resolution of a burst ACTT (ie. increase the range of possible values for the burst ACTT), but having a large number of cell streams to multiplex will reduce the effectiveness of the multiplier.

An example goal for the simulation experiments performed in this section could be to determine how the peak data transmission bit rate on the shared communications link varied with the number of potentially active callers. To satisfy this goal, the multiplexer was instrumented to report the ACTT value at its output port every 5 minutes of simulated "real time". This was achieved by getting the multiplexer object to repeatedly schedule `WAKEUP` messages for itself at 5 minute intervals. Upon receipt of the message, the current output port ACTT was written to a file and the next `WAKEUP` message was scheduled. Processing the `WAKEUP` message has no effect on the normal operation of the multiplexer.

A burst ACTT value can be easily converted into an actual bit rate. As 1000 units of the clock represent the time to send one cell at 155.52Mbs^{-1}, the current bit rate (B_R) can be calculated for a multiplexer output burst ACTT ($ACTT_B$) as follows:

$$B_R = \left(\frac{1000.0}{ACTT_B} \times 155.52\right) Mbs^{-1}$$

Figure 6.6 shows how the data bit rate on the transmission line varied, over a simulated period of eight hours, with the number of active **CBRsource** objects, for each of the experiments performed with the **BL2** simulator. If an experimenter worked for a telephone company, for example, and the model was of a residential backbone link, a conclusion could be that many more telephone subscribers could be supported before bandwidth limitations became an issue. Of course, the modeller would also have to have confidence in the telephone usage distributions used.

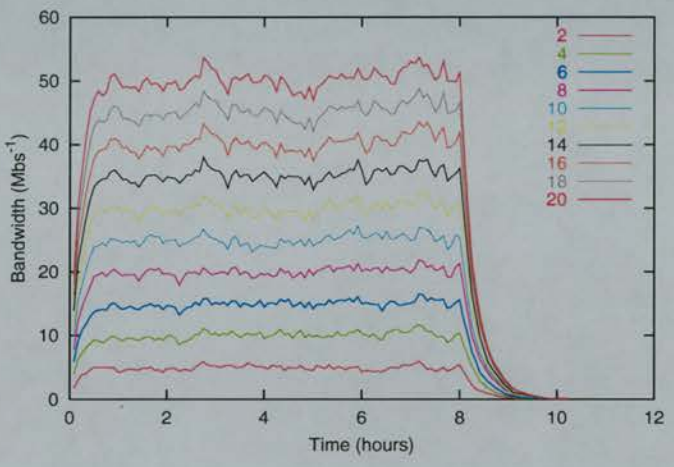


Figure 6.6: The total data transmission bit rate over time for each experiment with a varying number of **CBRsource** objects

6.3.3 Summary of section

The experiments performed in this section were designed to show how the burst-level techniques presented in this work responded to a high load scenario. It is clear that as the number of input channels to a multiplexer increases, the amount of work required to produce the appropriate output bursts also increases. The increase in work is seen as an increase in the simulation runtimes of the experiments performed. Detailed investigation of the bottlenecks inherent in the original core techniques lead to the revised techniques described in Chapter 4. The results in this section confirm that the use of the revised techniques can result in a dramatic decrease in the simulation runtimes observed for a high load simulation model. In the experiments simulating 20 active **CBRsource** objects, the performance of the **BL** simulator was close to the simulated time period (ie. taking 8 hours to perform the simulation), whereas **BL2** was of the order 6 times faster than "real" time.

The model simulated in the experiments in this section could also be simulated with a cell-level simulator (such as **CL** or **CLt** as produced in this work). However, the results of the burst-level simulations can be used to predict the scale of the runtimes for a cell-level simulation of the model. For example, the experiment involving 20 active **CBRsource** objects simulates of the order 3.5 billion cell transmissions across the fibre optic cable. Each **CBRsource** produces around 175 million cells, whereas each **CBRsink** receives approximately 175 million cells. Performing this experiment with the CL simulator would require at least 10 billion events to be processed, which would take around 10 hours if 300000 events could be processed per second (as was the case for **CL** in Section 6.2.3.7). Such a runtime would be longer than the simulated time period (ie. 8 hours). Estimated **CL** runtimes are shown in Figure 6.5 for each experiment performed with the burst-level simulators.

In terms of the accuracy of the results produced, both burst-level simulators were shown to produce decreasing average burst ACTT results as the number of concurrent sources increased. This was as expected, due to the large number of streams diminishing the effect of the ACTT multiplier which was introduced to increase the resolution of burst ACTT values, and hence improve their accuracy. Choosing an even larger ACTT multiplier would improve the accuracy of the techniques, but careful thought would have to be made about the size of the integer shift constant used in the **BL2** simulator. The choice of the shift constant, relative to the size of the ACTT values used in a simulation, has a direct bearing on the accuracy of integer division by reciprocal multiplication. Choosing a shift constant which is too large will increase the likelihood of integer overflows, even for 64-bit values. The clear performance benefits of the **BL2** simulator for high loading scenarios make it the ideal choice in these situations, but care must be taken to ensure sufficient accuracy of the results is obtained.

6.4 Very high speed networking example

The aim of this example simulation model is to show how the integer-time burst-level simulation techniques can be used to model very high speed communication links. This model differs from the previous examples in this chapter as the fundamental notion of what a burst represents is altered to model a different transmission quantity. If the notion of a burst can redefined in this way, the simulation environment and objects can be used to model networks other than the cell-based networks for which they were designed. This broadens the appli-

cability of the efficient simulator, and the simulation objects, for use in other communication modelling studies.

Rather than consider an ATM-network cell to be the minimum unit of transmission, the model in this section considers the byte (ie. one *octet* of bits) to be the lowest level considered. In this case, a burst represents a *packet* of bytes transmitted at the same data rate. The divergence from the usual notion of a burst as described in this work, is that no inter-byte gap is considered in each burst of bytes (ie. unlike the the inter-cell arrival time which is included in the ACTT value for a burst of cells). The ACTT for a byte represents the transmission time for that byte with no extra delay, and a burst represents a continuous stream of bytes.

The inspiration for this model is the British Telecom *SynchroLan*[27, 50] optical time-division multiple access (TDMA) local area network technology demonstration. SynchroLan was used as the driving technology for the Edinburgh Configurable Optical LAN Environment (ECOLE)[18] project, which is examining the challenges of enabling workstations to utilise very high speed guaranteed bandwidth local area networks. Recent advances in personal computer high speed data interfaces (particularly the *Accelerated Graphics Port* (AGP)[61]) mean that low latency interfaces to very high speed networks are now possible.

The SynchroLan demonstration consisted of a fibre optic local area network comprising 16 separate data channels each with a constant bandwidth of 2.5Gbs^{-1} (making an aggregate bandwidth of 40Gbs^{-1}). Each workstation node (up to a maximum of 16 nodes) on the SynchroLan had separate read and write interfaces to the fibre optics, with the write interface permanently "tuned" to one of the 16 available channels. A workstation could "tune in" to any of the 16 channels to receive data from another node. Out-of-band signalling (over a standard 10Mbs^{-1} Ethernet) was used to change the read channel for each workstation node. Each communicating pair of nodes had a guaranteed bandwidth of 2.5Gbs^{-1}, irrespective of the communication status of other nodes on the network. Broadcasting and multicasting capabilities were supported by allowing more than one workstation read node to "tune in" to the same workstation write node, each receiving data at 2.5Gbs^{-1}. Although each channel could use the 2.5Gbs^{-1} data rate, the original BT demonstration network used off-the-shelf ATM network interface cards (NIC) to limit the bandwidth of each channel to 155Mbs^{-1}. Each data bit transmitted by a write node was actually represented by 16 pulses on the 2.5Gbs^{-1} channel to scale the 155Mbs^{-1} data rate to the SynchroLan channel. Each node read interface sampled 16 pulses from SynchroLan to provide one input data bit to the

network node.

One of the research goals for the ECOLE project is to examine the impact of a low latency, high bandwidth interconnection network on the performance of a *Network Of Workstations* (NOW) using the *Distributed Shared Memory* (DSM)[87] model. In the DSM model, the memory located in each node can be accessed by each other node in the network as if it was part of one shared memory pool. Ensuring the consistency of the memory contents located at each node is maintained through the passing of messages to indicate any changes in the information stored. Traditionally, the use of a legacy network (such as Ethernet) as the interconnection network has been the bottleneck in a NOW cluster. Various techniques such as *lazy release consistency checking*[67] have been used to reduce the message passing overhead and thus increase the performance of DSM system. The provision of a high bandwidth interconnection network with low message latencies, is one means of helping to improve the performance of a NOW cluster using the DSM model.

6.4.1 The model

Figure 6.7 shows the object structure of the simple NOW cluster model used in the experiments in this section, the details of which are explained in the following few pages. The interconnection network is a simplified model of the SynchroLan network which ignores the need to "tune" each node into the write channel of its communicating partner. Each workstation is also a simplified model of a contemporary personal computer equipped with both *PCI* (Peripheral Component Interconnect) and *AGP* bus interfaces. Each network interface card (NIC) connecting a workstation to SynchroLan is modelled as separate input and output channels (equivalent to the SynchroLan read and write interfaces).

As a simplification, the SynchroLan model does not use any signalling to establish communicating pairs of nodes. Therefore it is possible that more than one node may send data to the same workstation concurrently. This is impossible with SynchroLan, as a node can only tune in to the write channel of one other node. The use of a multiplexer at each input port ensures that multiple arriving burst messages are handled correctly. For clarity in Figure 6.7, the connections between each output demultiplexer and input multiplexer are shown as multicast links. In the simulation model, each demultiplexer has a unique output port connected to each multiplexer (apart from the one acting as its own input port).

Each workstation object has a very simple operation. When the simulation starts, a workstation simulates a period of computation which lasts for a random

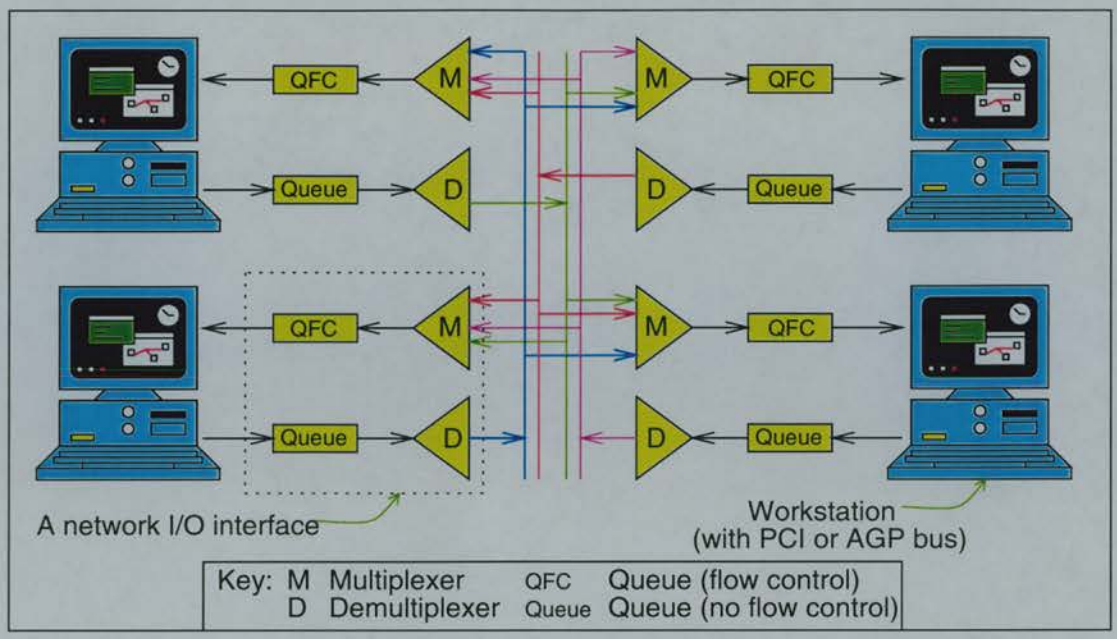


Figure 6.7: The object model used for the very high speed networking experiments

number of clock cycles. The length of the computation time is drawn from a negative exponential random distribution. At the end of the computation period (which is signalled by the arrival of a `WAKEUP` message at the object), the workstation sends a *page* of bytes to another workstation chosen at random (using a *Randint* distribution). The transmission of a page represents the workstation sending a data consistency update message to another workstation. The page is a convenient unit, as it represents the virtual memory structure of a modern computer. For this class of workstation, page faulting as reported by the virtual memory controller, can be used to provide the impetus for the creation of consistency update messages in a DSM model (also known as DVSM (Distributed Virtual Shared Memory) in this case).

Once a page has been transmitted, the workstation object simulates another random length period of computation. At the end of every computation phase, a page message is generated for another workstation chosen at random. This cycle of computation, followed by the generation of a page message continues until the workstation object receives a `TERMINATE` message from the workstation object set to be the *Master*. One workstation object is set to be the Master and the other workstations in the cluster are set to be *Slaves*. The Master determines the length of each simulation run by executing a fixed number of computation/page message generation cycles. Once the required number of cycles has been completed, the Master schedules a `TERMINATE` message for each of the Slaves. The effect of the arrival of a `TERMINATE` message at the Slave is to cease the computation/page

message generation cycle, but to complete the current operation at the slave before finishing.

The page is chosen as the minimum burst of bytes to be transmitted in a synchronisation message. Each virtual memory page in an IBM-compatible PC is 4096 bytes, hence each burst generated is 4096 bytes in length. The use of the byte as the "currency" of a burst replaces the cell for this model. Each burst is still represented by a pair of `START` and `FINISH` messages, and the duration of the burst is equal to the product of the number of bytes contained and the time to transmit one byte (the ACTT in this case). The burst ACTT is the reciprocal of the data transmission rate for the particular bus modelled.

The SynchroLan network interface card (NIC) in each workstation is modelled using four objects. A queue with flow control object (see Section 3.6.4 on page 70) and a multiplexer represent the input channel on the card, whereas a queue without flow control and a demultiplexer represent the output channel. Either the workstation PCI bus or the AGP bus may be used to send and receive data to SynchroLan. Each bus is modelled as a resource which may be in one of three states; idle, sending data or receiving data. The length of the send or receive states on the bus is determined by the bus speed and the length of the burst being transmitted across the bus.

The generation of a page message by a workstation involves seizing the bus (ie. putting the bus into the send state which lasts for the duration of the burst transfer to the output queue) and sending a burst to the output queue. The output queue then passes the burst to the demultiplexer which routes the burst to the input multiplexer for the intended destination node. As neither the output queue nor output demultiplexer schedule messages for themselves, the event list can be bypassed for sending bursts between the workstation, output queue and output demultiplexer objects. The messages generated by each demultiplexer for each of the multiplexer input nodes are inserted into the global event list as normal to ensure the correct time ordering of events.

The receipt of a page message begins when the input multiplexer sends an arriving burst to the input queue. When the queue receives the burst `START` message, a `QREQ` message is sent to the workstation object alerting it that data is waiting in the input queue. If the workstation object is in the computation phase, a reply `QOK` message is generated immediately to initiate transfer of the burst from the input queue to the workstation. When this is occurring, the bus goes into the receive state for the duration of the burst transfer (governed by the length of the burst and the bus data rate). When the workstation is sending data

to the output queue via the bus, no data from the input queue can be transferred to the workstation, hence the use of a queue with flow control as the input queue. When the workstation is signalled by the input queue that data is waiting, the next time the interface bus goes idle (ie. at the end of the send phase), the input queue is passed a `QOK` message to start data transfer.

The choice of workstation data bus (PCI or AGP) and the state of the bus during the sending or receiving of bursts is also a simplification. When the workstation is in the computation state, the arrival of a `QREQ` message from the input queue pauses the workstation for the duration of the burst transfer over the bus. The workstation resumes the computation state (for the time period remaining when the `QREQ` was received) before sending a message burst. In the same way, the generation of the output page burst is exclusive, for the time required to transfer the burst across the bus to the output queue. In an *IBM compatible PC*, for example, *Direct Memory Access* (DMA) would be used to enable the SynchroLan NIC to pass data over the chosen bus to and from main memory without the intervention of the processor. This allows the processor to continue computation even when the data bus is busy. The model used in the experiments does not explicitly model the use of DMA transfers, but the processor may be considered to be processing data during the bus I/O phase. Thus a precise characterisation of the number of compute cycles is as the number of non-overlapped compute cycles; however, this does not affect the discussion of the experimental results. In a true DSM system, workstations would be required to wait for *barrier synchronisation* messages), for example, and this would affect the time spent performing useful computation. At the end of a simulation run, the percentage of time spent by the workstation in the computation phase and performing I/O over the bus are reported.

Table 6.12 gives the integer clock conversions necessary for the parameters modelled in each simulation in this section. The PCI bus is assumed to be able to transfer data at a rate of 132Mbyte s^{-1} (ie. the maximum permitted), whereas the AGP bus is assumed to transfer data at a rate of 528Mbyte s^{-1}. When the PCI bus is being used, the SynchroLan NIC is assumed to be the only device on the PCI bus. In an IBM compatible PC, the PCI bus would be shared by other peripherals such as graphics cards, Ethernet cards, and any other expansion device. The AGP bus is specified as an independent bus for single devices, and so does not require the same simplification. Each SynchroLan channel is assumed to operate at 2.5Gbs^{-1} rather than the 155Mbs^{-1} as demonstrated in the original prototype. Performing a simulation study with the reduced bandwidth would

be an interesting experiment for the ECOLE project, but the aim of this simple experiment is to show the simulation environment modelling very high speed links. The processor in each workstation is assumed to have a clock speed of 600MHz which is relatively modest by current standards.

Transmission speeds: Value	Integer clock units	
	1 bit	1 octet (byte)
AGP (528Mbyte s^{-1})	1000	8000
PCI (132Mbyte s^{-1})	4000	32000
SynchroLan (2.5Gbs^{-1})	1689	13512
1 CPU clock (600MHz)		7050

Table 6.12: The conversion of "real world" metrics to integer clock units for each simulation

6.4.2 The experiments

Four sets of experiments were performed using the model described in the previous section, where the parameter varied was the mean number of clock cycles each workstation spent in the computation phase. For each mean computation length, the network model was run in two configurations. In the first configuration, the PCI bus was used to connect the workstation to SynchroLan. In the second configuration, the AGP bus was used to connect the SynchroLan NIC to the workstation. Five repetitions of each experiment were performed with the random number seed altered for each experimental run. Table 6.13 shows the experimental parameters used for the experiments. All of the experiments were performed on the same workstation, with care taken to ensure that it was unloaded by other user processes.

Value	Parameter
No. of compute/ produce output page cycles	100000
Page (burst) size	4096 bytes
Input queue size	64Kbyte
Output queue size	64Kbyte
Average no. clock cycles in computation phase	10000, 100000, 1000000 & 10000000

Table 6.13: The experimental parameters

The results recorded for each experiment were as follows (an example of a simulation result report generated for one experimental run can be seen in Section B.2 in Appendix B):

- The total simulation runtime.
- The simulated program runtime in the experimental run (as reported by the Master workstation object). This is a measure of how long the NOW cluster takes to perform the computation requiring the number of computer/output page generation cycles specified.
- The average percentage of time each workstation spent in the computation phase and in the I/O phases (ie. sending and receiving bursts over the bus used in the simulation run). The average for each experiment was taken over the results for the four workstations.
- The average SynchroLan communication channel utilisation. Each demultiplexer object was instrumented to record the total time each workstation spent sending data over SynchroLan. The average result produced in each experiment was calculated from the four workstation results.

As five repetitions of each particular experiment were performed, average values could be computed for each result along with a calculated 95% confidence interval.

6.4.3 The results

The mean number of clock cycles spent in the computation phase determined the NOW cluster runtime simulated in each experiment, whereas the fixed number of compute/output page generation cycles at the Master workstation governed the communication to simulated runtime ratio. Table 6.14 shows the average simulation experiment runtimes and the average simulated NOW cluster runtime values for each experiment (shown with calculated 95% confidence intervals).

Bus type	PCI		AGP	
Av. compute (clock cycles)	Expt. runtime (secs)	Sim. NOW runtime (secs)	Expt. runtime (secs)	Sim. NOW runtime (secs)
10000	39.52	8.14 ± 0.02	52.52	3.75 ± 0.02
100000	38.39	23.53 ± 0.09	46.67	19.03 ± 0.09
1000000	37.79	173.79 ± 0.86	45.59	169.14 ± 0.74
10000000	37.66	1674.26 ± 7.74	45.43	1669.45 ± 7.55

Table 6.14: The average experiment runtimes and simulated times for each experiment

As would be expected, the simulated NOW cluster runtime in each experiment increased with the average number of CPU cycles spent in the computation phase. As would also be expected, the AGP bus experiments reported shorter simulated NOW cluster runtimes, due to the faster bus transfers of bursts between each workstation and SynchroLan. The difference made by the faster I/O is more apparent when the communications to processing ratio is high, as in the 10000 CPU cycle experiment. When the communication to processing ratio decreases, the improvement in the simulated NOW cluster runtimes due to the use of the faster I/O bus is decreased.

In terms of the average runtimes for each simulation experiment, the PCI configuration is faster than the AGP configuration. This is due to a greater number of events being processed in each AGP simulation due to the workstation bus being faster than each SynchroLan channel. This requires more work from each queue object, especially when each input queue has to deal with multiplexed burst arrivals. However, both simulators offer better than simulated runtime performance when the communications to processing ratio decreases in the experiments.

Table 6.15 shows the average percentage of the simulated NOW runtime spent in the computation and bus I/O phases. As would be expected, the AGP bus experiments spend a lesser percentage of the simulated NOW cluster runtime performing I/O over the bus. This is one of the factors leading to the reduced NOW cluster runtimes for the AGP configuration (when compared to the PCI configuration) seen in Table 6.14.

Bus type	PCI		AGP	
Av. compute (clock cycles)	% time in computation	% time in bus I/O	% time in computation	% time in bus I/O
10000	20.47 ± 0.03	79.53 ± 0.03	44.38 ± 0.06	55.62 ± 0.06
100000	70.85 ± 0.04	29.15 ± 0.04	87.63 ± 0.04	12.37 ± 0.04
1000000	95.95 ± 0.02	4.05 ± 0.02	98.56 ± 0.02	1.44 ± 0.02
10000000	99.58 ± 0.004	0.42 ± 0.004	99.85 ± 0.005	0.14 ± 0.005

Table 6.15: The average experiment runtimes and simulated times for each experiment

Table 6.16 shows the average communication channel usage for each of the experiments. Both sets of experimental results show that regardless of the bus used for the SynchroLan data transfers in the workstation, it is very difficult to

fill the large bandwidth offered by each channel. However, especially in the high communication to processing ratio experiments, the SynchroLan bus utilisation is higher in the AGP experiments, due to the higher bus bandwidth offered by AGP. The SynchroLan average channel utilisations for each bus type are almost identical when the communications to processing ratio decreases

Bus type	PCI	AGP
Av. compute (clock cycles)	% channel utilisation	% channel utilisation
10000	16.0739 ± 0.0696	34.8150 ± 0.3196
100000	5.5608 ± 0.0322	6.8762 ± 0.0476
1000000	0.7527 ± 0.0042	0.7729 ± 0.0047
10000000	0.0781 ± 0.0005	0.0783 ± 0.0005

Table 6.16: The average SynchroLan channel utilisations in each experiment

Although the model simulated in these experiments was very basic and introduced many simplifying assumptions, it is clear that the transfer of individual virtual memory pages in a DSM framework results in small channel utilisations on SynchroLan. Increasing the communications to processing ratio does not introduce a penalty when a high bandwidth interconnection network is used along with a fast data bus in the workstation. If features such as DMA transfer of information between the main memory and the interface card were used, and larger packets of information were sent at each message transfer, even better performance might be seen.

A real DSM network of workstations would exhibit more complicated behaviour than the example presented in this model. However, this model shows the limits of what might be achieved if protocol overheads are ignored and the workstations are able to communicate freely with each other.

6.4.4 Summary of section

The aim of the experiments in this section was to show how the integer-time burst-level techniques and objects presented in this work can be used to model very high bandwidth communication links. A very simple model, based on a real example of an existing high bandwidth LAN, was used to demonstrate this. The model, and the experiments performed, showed the applicability of the burst-level techniques for a setting other than a cell-based network such as ATM. Any

communicating system, where the basic unit of transmission can be modelled as a burst of some other basic entity, can be simulated with the techniques presented. The ACTT value in each burst can be defined to model a transmission time plus and interarrival time, or just a transmission time to make the burst represent a *continuous* burst of the lower level entity. The simulation objects produced in this work will queue, multiplex and demultiplex bursts regardless of the abstraction represented by the burst in the simulation model.

Although the model was very simple, the results showed that the simulation experiment runtime depended on the communications to processing ratio encountered within each experiment. The higher the ratio, the more work the simulator must perform, and this leads to the simulated NOW cluster runtime being less than the actual experimental runtime (ie. worse than real time performance). Better than real time performance was seen when the communication to runtime ratio was decreased.

Altering the model to communicate with larger burst sizes (in bytes) or to exhibit more complex behaviours (such as out-of-band signalling to create communicating pairs of nodes or DMA bus transfer) would also have an influence on the experimental runtime verses the simulated NOW cluster runtime.

6.5 Summary of chapter

The aim of this chapter was to show how the simulation objects presented in Chapter 3, and refined in Chapter 4 could be used to model three different scenarios. The first scenario presented a "typical" communication network model which involved many interconnected simulation objects. The second scenario showed how the integer techniques coped with a high load situation where many individual cell streams had to be multiplexed. The final scenario was that of modelling very high speed communication links as might be found in an optical local area network.

The results from the first experiment, the simple network model, showed that the burst-level simulators are capable of providing good runtime speedups over the equivalent cell-level simulations. The use of the improved integer techniques did not enhance performance due to the the low levels of multiplexing encountered. However, the improved integer techniques did help to reduce the magnitude of delays introduced by the burst-level multiplexing technique. The magnitude of the delays was found to be related to the number of simulation objects traversed, as well as the relative ACTT values of the bursts considered. It was found that

having well-matched ACTT values in a simulation model also reduced the scale of errors. Burst-level queueing was found to give good agreement with the cell level results as regards the scale of cell loss over a simulation run. The individual cell stream allocations of the loss were found to agree less, with more loss being apportioned to faster streams. Examination of the delays incurred in the multiplexing technique, as well as the burst splitting used to apportion cell loss over the streams in a queued burst were recommended as areas for future study.

The results from the second experiment, the mass multiplexing model, clearly demonstrated the advantages of using the revised integer techniques when many cell streams have to be merged. The original integer techniques were shown to give dramatically increasing experimental runtimes as the number of active streams to merge increased. The revised techniques also showed increased runtimes as the number of streams increased, but to a much lesser extent. The accuracy of the burst ACTT values was shown to drop as the number of streams increased. This was as expected, due to the beneficial effects of the ACTT multiplier being reduced by the large stream count. When using the revised integer techniques, it is important that the fixed integer shift constant (used to calculate left shifted reciprocals) is kept as small as possible to help reduce the likelihood of integer overflow. Using large ACTT multipliers to help improve ACTT accuracy will reduce the accuracy of integer division by multiplication of reciprocals if the integer shift constant is too small. Careful selection of the integer shift constant, and the size of the ACTT multiplier, must be made to ensure sufficient result accuracy when using the revised integer techniques.

The results from the third experiment, the very high speed network model, showed that the extent of communication simulated within the model affects the experimental runtimes. If a very high speed network simulates many burst transmissions, the simulator will be required to perform lots of work. If the simulated transmissions model a short period of "real" time, the simulation will give performance which is worse than real time. However, if the extent of communication to "real" time in the model is reduced, the burst-level simulator can give better than real time performance. The complexity of the simulation model, as well as the extent of simulated communication, will have the greatest bearing on the runtimes observed. One strength of simulation is the ability to model and observe behaviour which is difficult to observe in reality, and the use of burst-level simulation for very high speed networks can be viewed as one such tool.

Chapter 7

Conclusions and future work

7.1 Overview

The aim of the work presented in this thesis was to propose a set of novel techniques to improve the experimental runtimes of simulations of very high speed communication networks. Two main approaches were used to help achieve this aim. The first approach was to abstract the level of detail from the cell to the burst level. The second was to harness the implicit performance advantage of integer arithmetic in modern microprocessors when designing the algorithms used in the techniques. The use of an integer variable to model time is an essential part of such a design approach. The importance of an efficient simulation environment with which to perform each simulation was stressed, and a bespoke C++ environment with a choice of event list management schemes was presented. The aim of complete simulator code portability was achieved by using standard C++ throughout the development of the simulation environment and by not relying on any system-specific features (eg. such as a threads package).

Simulation techniques, along with simulation objects to implement each technique, were described to provide the fundamental simulation operations of burst multiplexing, queueing, demultiplexing and switching. Analysis of the algorithms highlighted two core operations which could be further optimised to help improve overall simulation runtimes. Each revised core technique was shown to improve runtime performance, while maintaining the accuracy achievable with the techniques.

Each burst-level simulation object was tested for performance and accuracy issues against an efficient cell-level bespoke C++ simulator also developed in this work. The cell-level simulator was capable of processing between 300000 and 500000 events per second (depending on the simulation model) on the workstation used to perform all of the experiments undertaken in this thesis. The burst-level

simulators were capable of executing around 100000 events per second depending on the configuration of the model being simulated.

The simulation objects described were designed to form the building blocks for simulation models of communication networks. To illustrate this, three simple example simulation models were presented, each using the integer-time burst-level simulation techniques to model a different scenario. The third example, in particular, highlighted the appropriateness of the techniques for use with simulation models other than the cell-based communication networks for which they were designed.

The conclusions from the work presented in each chapter are summarised in the following sections.

7.2 Conclusions

7.2.1 Burst-level simulation

Chapter 3 introduced the burst-level techniques and the simulation objects to implement them. An efficient bespoke C++ simulation environment was presented which could use either an indexed doubly-linked list or a post-ordered tree global event list structure. The pros and cons of each event list structure were outlined, with the conclusion that the simulation time distribution of event insertion is critical to the performance obtainable.

The notion of a burst (as defined in this work), and how a burst relates to the integer clock, was described. Abstracting the level of detail to the burst level reduces the total number of events which need to be processed in a simulation, as a burst represents a group of cells. A cell-level simulator must schedule events for the transmission of individual cells and so requires a great deal of compute time. The granularity of burst sizes has a bearing on the accuracy obtained in the burst-level simulator, as well as on the work necessary to process bursts in the simulation objects. Too large a burst size will "average" interesting behaviour out of the model, whereas too small a burst size will require the simulator to perform more work.

Strict rules for what constitutes a burst in this work were presented. It was found that such rules were necessary to design and implement objects to provide the core operations for burst-level simulation.

7.2.2 Core burst-level techniques

Chapter 4 presented and analysed two core techniques used in the burst-level simulation objects described in Chapter 3. The two techniques were *burst creation* and *burst splitting*. It was found that the inclusion of an *ACTT multiplier* to improve the range of burst ACTT values, and hence their accuracy, was desirable. The larger the ACTT multiplier used, the better the accuracy obtained. The impact of using ACTT multipliers on the runtime performance of each core technique was found to be minimal.

Once the core techniques had been identified and analysed, the next step was to further optimise them by examining the integer instructions executed by each. The integer divide operation was found to be the most costly in terms of throughput and latency, and so modified versions of the core techniques which replaced the divide instructions were presented. Each revised core technique was analysed, with the conclusion being that runtime performance was improved through their use, but not at the expense of the accuracy of each technique.

7.2.3 Accuracy and performance of the simulation objects

Chapter 5 compared the accuracy and performance of each burst-level simulation object (using both the original and revised core techniques) against equivalent cell-level simulation objects. In summary, the following conclusions were made about the runtime performance observed for the burst-level techniques.

1. The burst-level simulation objects offered good speedups over the cell-level equivalents.

2. The average burst size was found to have the greatest influence on the runtime performance of each burst-level simulator. The larger the average burst size, the better the performance.

3. The time density of burst arrivals at the multiplexer also influenced the runtime performance. The greater the time density of concurrent burst arrivals, the greater the degree of burst fragmentation that occurs. The number of bursts to be processed in a simulation is related to the degree of burst fragmentation, and so the greater the time density of burst arrivals at a multiplexer, the lower the performance.

4. On average, the indexed doubly-linked event list has more predictable behaviour than the post-ordered tree. The post-ordered tree structure was

shown to incur large performance penalties for certain experiments performed in this chapter. The performance of the indexed event list depends on the maximum number of events pending in a simulation versus the maximum number of events described by an event index. Choosing the maximum number of events per event index is important, and insight gleaned from previous simulation runs can be used to aid this task.

5. Bypassing the global event list can be used to help improve the runtime performance. In certain circumstances the locality of interconnected simulation objects can be exploited by making one object the slave of the other. Rather than schedule events for the slave via the global event list, the master can pass them directly to the slave via a procedure call (ie. a method in the slave object class definition). This technique reduces the overhead of event scheduling and so improves performance.

6. The revised core techniques give an increasing performance benefit as the number of cell streams to multiplex increases. If little burst multiplexing exists in a simulation model, the revised integer core techniques make little difference to the runtime performance observed. However, if the number of cell streams to multiplex is large, the benefit increases with the number of streams.

The following conclusions regarding the accuracy of each of the simulation techniques were made.

1. The size of each burst (in cells) was found to influence the accuracy of the final ACTT value for each cell stream. The larger the burst sizes simulated, the more accurate the final burst ACTT values for each stream. The revised integer techniques were found to *underestimate* burst ACTT values, and so produce underestimates of final cell stream ACTT values.

2. Each simulation object was found to introduce implicit delays to bursts passing through it. The multiplexer was found to exhibit the worst delays, which increased in magnitude with increasing time density of input burst arrivals. The delays are caused by accumulated delays at the multiplexer output port due to aligning bursts to cell transmission boundaries, and also due to inaccuracy in the burst ACTT values. The underestimated burst ACTT values produced when the revised integer techniques were used were found to reduce the scale of the delays introduced in the burst-level objects.

3. The burst-level queue objects were found to be good at estimating cell loss at the macroscopic scale (ie. over an entire simulation run), but to show variation in the allocation of the cell loss to each cell stream passing through the queue.

4. Zero cell length bursts (which waste effort in the simulator) and overlong bursts (which affect the accuracy of burst ACTT values) are side-effects of the burst-level techniques. Adding scheduling delays to object output ports, as well as adding output port queue objects, was found to reduce the number of zero cell length or overlong bursts encountered in a simulation run.

7.2.4 Experimental performance

Chapter 6 presented three simple experiments showing the integer-time burst-level simulation techniques used in different modelling scenarios.

In the first example, a simple ATM-like cell-based communications network was modelled. Each of the burst-level simulators was found to offer good runtime speedups when compared with equivalent cell-level simulations. As was observed in Chapter 5, each burst-level simulation object introduced a delay to bursts traversing the object. The number of objects traversed by a burst, as well as the relative ACTT values for bursts multiplexed onto the same link, were found to influence the magnitude of the delays observed. The burst-level simulator using the revised core techniques offered lower delays due to the burst ACTT underestimation inherent in the techniques. However, the scale of multiplexing encountered in the model meant that no speedup gain was seen over the simulator using the original integer techniques. Cell loss was observed in the experiments, and the burst-level queue was found to be in good agreement with the cell level results as regards the scale of the loss observed over an entire simulation run. Examination of the cell losses allocated to individual streams showed that streams with lower ACTT values suffered disproportionate cell loss ratios.

In the second example, each burst-level simulator was tested in a scenario where the number of individual cell streams to multiplex was large. As the number of streams increased, the performance of the burst-level simulator using the original integer techniques got progressively worse. The simulator using the revised integer techniques also suffered a performance drop as the number of streams increased, but offered substantially better performance than the simulator using the original techniques. This was as expected, as the revised integer techniques were found to give increasing benefit as the number of streams increased (see

Chapter 4). Both burst-level simulators saw a drop in the accuracy of the final burst ACTT for each stream as the number of streams increased. This was due to the beneficial effect of the ACTT multiplier being diminished by the large number of streams. The choice of a large ACTT multiplier must be reconciled with the choice of the fixed integer constant used in the revised integer techniques. The simulator using the revised integer techniques is recommended for models involving a high degree of burst multiplexing due to the runtime advantage offered. However, to ensure acceptable accuracy of the results, care must be taken when choosing the ACTT multiplier and a suitable integer shift constant for calculating left shifted reciprocals. Poor selection of these parameters may lead to inaccurate integer division calculations, or integer overflow, even when using 64-bit variables.

The third example showed how the simulation environment produced, and the integer-time burst-level simulation objects, could be used to model a network which was not based on the transfer of cells. Instead, each burst represented a *packet* of bytes, where no inter-byte delay was modelled within the packet. The results showed that the scale of the communication to simulated time ratio in each model affected the runtime performance of the simulator. Issues such as model complexity will also have a bearing on the performance observed.

7.3 Future work

7.3.1 Integrating object behaviours

The experiments in Chapter 5 showed that bypassing the global event list to link two (or more) simulation objects could further reduce simulation runtimes. Exploiting this object locality was also used (where possible) in the simple example experiments in Chapter 6 to maximise the performance of the burst-level simulators. The next step would be to merge the operations of the objects capable of being joined in this way into individual simulation objects. In the case of the multiplexer object driving a slave queue without flow control, the integration of the queueing operation into the multiplexer would present few challenges. This would also be the case for a master queue object connected to a demultiplexer slave. Producing single objects with integrated behaviour would help to reduce the overhead incurred by using the compound object. This benefit would be at the expense of code complexity of the merged objects.

The compound burst-level switch objects used in the experiments in this work would be more difficult to implement as single objects. Event list bypassing *cannot* be used to link each input demultiplexer to each output multiplexer, because

of the need to ensure the correct time ordering of burst arrivals. However, demultiplexers and multiplexers with integrated queueing could be used to provide input and output buffering as required. An ideal goal would be to produce a fully integrated switch object to replace the collection of multiplexer, demultiplexer and queue objects needed to provide a compound burst-level switch. Such a switch object would be fairly complex to produce, but may give an excellent runtime advantage due to the reduction of the event traffic necessary for the operation of the compound switch.

7.3.2 Minimising multiplexer delays

The experiments in Chapter 5 and Chapter 6 highlighted the large accumulated burst scheduling delays which can result from the actions of the multiplexing process. The use of the revised integer techniques was shown to reduce the magnitude of these days, as was having well-matched stream ACTT values in a simulation model. Other factors, such as the time density of input burst arrivals to a multiplexer object, also have an influence on the delays observed.

The delays which can accumulate at the multiplexer output port result from the need to align output burst messages to cell transmission boundaries and the inaccuracy of burst ACTT values. When an incoming burst message is made to wait behind a delayed output burst, cells from the input burst effectively accumulate within the multiplexer when they should be in an output burst. This will increase the output delay, as the burst creation process used to generate the next output burst (which includes the new input burst) ensure that the number of cells included from the burst reflects the original arrival time of the burst at the multiplexer (ie. including the delayed cells from the burst). Scheduling an output burst which includes this number of cells adds to the accumulated delay.

One possible means of minimising the delay experienced by newly arriving bursts could be to split output bursts whose `FINISH` message time is greater than the simulation time of the new `START` message arrival. The output `FINISH` message is generated when the state of an input changes (ie. the arrival of the new burst `START`), and so the split could be performed before the generation of the new output `START` including the newly arrived burst. In effect, this would limit the delay to the new burst, but would alter the cell stream composition of the previous, and new, output bursts. The "remainder" of the split output burst would have to be merged with the new stream. The multiplexer would then have to schedule a message for itself to ensure that the remainder of the split output burst is sent in its entirety. Studying this modification, and its implications on

the multiplexing operation, would be an area of further work. Any alteration of the microscopic stream structure of bursts will also influence the behaviour of the burst-level queue. Attempting to reduce the message delays added to bursts passing through a multiplexer may also influence the cell loss allocation between cell streams in the burst-level queue.

7.3.3 Implementing priority queueing

The burst-level queueing techniques presented in this work are designed to allocate cell loss fairly among the component cell streams in a burst. The allocation of loss is based on the ACTT of each stream and the number of cells carried. Real network buffers usually provide some sort of *priority queueing* algorithm which will selectively discard cells based on some criteria. In an ATM cell, for example, a *Cell Loss Priority* (CLP) bit in the header can be set to mark the cell as a candidate for loss if cells need to be discarded from a buffer. Buffer implementations in networking hardware typically examine cell loss priority options for dropping cells when the buffer occupancy reaches a certain threshold.

Such a priority-based cell discard scheme, based on the buffer occupancy reaching a predetermined threshold, could be added to the queue objects described in this work. Each queue object would have to monitor the status of the buffer occupancy, and perform a burst split at the time when the threshold is reached. The cell streams described in the current burst could then be examined, with cell loss applied to each stream depending on the cell loss priority scheme implemented. The remainder of the burst would then be sent from the queue output port with the stream cell counts reflecting the priority-based cell loss applied to the burst.

Using buffer thresholding may also help to improve the allocation of cell losses to individual cell streams in multiplexed bursts. The experiments in Chapter 5 and Chapter 6 showed that the burst-level queues were accurate at the macroscopic level, but could show variation in the allocation of cell loss between component streams.

7.3.4 Parallelising the techniques

As the simulation techniques presented in this thesis are based on abstracting the behaviour to the burst level, there is scope for parallelising the techniques. This is because there is a low communication (ie. burst message generation) to processing (ie. processing each burst message in an object) ratio inherent with modelling at the burst level. The use of *Parallel Discrete Event Simulation* (PDES) could be particularly beneficial when simulating very large networks. The classic problem

with PDES is dimensioning a simulation model among Logical Processes (LP) to ensure fair load balancing. A good candidate for where to place a logical split between LPs would be a shared communications link (such as one of the inter-switch links in the network experiment in Section 6.2 on page 201).

The use of a TimeWarp (see Chapter 2) simulation kernel could be considered, with versions of the simulation objects created which could have their states "rolled back" if an out of time order message was received by the LP. Altering the simulation objects to have this capability would require a fair amount of work. Care would also have to be taken to ensure that a suitable synchronisation mechanism was chosen so that excessive state roll backs did not compromise the speedup offered by using a parallel approach or the accuracy of the sequential versions of the techniques. For example, Bocci[15] examines these issues when considering a parallel version of Pitts' cell rate method[91] (also an abstracted technique), and finds that the performance and accuracy of the technique are compromised unless careful thought is paid to parallel synchronisation issues.

7.3.5 Enhancing the simulation environment

The simulation environment and the simulation objects produced in this work are prototypes. Further optimisation of the simulation objects (above and beyond the behavioural integration proposed in Section 7.3.1) may lead to even better performance. Careful optimisation of data structures and algorithms, as well as the use of a good optimising compiler, will benefit the runtime performance of the code when it is optimal for the target architecture of the host computer.

In the current burst-level simulators, the onus is on the modeller to ensure that the top level simulation code correctly instantiates and configures all of the objects necessary to perform a simulation. Such a process can be prone to error, especially when producing a large network model. Integrating the simulation environment into a visual tool capable of automating the model construction would be advantageous. For example, Fahmy[36] presents a graphical tool which can produce and perform network simulations, while ensuring that errors are not introduced into the model. Alternatively, the *HASE*[24] (Hierarchical computer Architecture design and Simulation Environment), developed in Edinburgh, provides an object-oriented simulation environment for model design and execution. Integrating the simulation objects and techniques described in this thesis would be straightforward, but integration of the simulation engine (to replace the system provided by HASE) would be more difficult. As an alternative to producing a graphical front-end for model design, the Italian CLASS team provide a high-level

network description language called *RC*[76]. RC parses the high-level model and produces the C code necessary to simulate the model in the CLASS simulator. Future work could address the use of the simulation framework developed here as an efficient back-end to such tools.

7.3.6 Summary

The future work proposed in the preceding sections highlights some interesting directions in which to develop the techniques. The results presented in this thesis show that the techniques provide a strong foundation for improving the simulation runtimes of models describing very high speed communication networks. The suggested future work would lead both to enhanced performance and accuracy, and to enhanced useability.

Appendix A

Experimental results for Chapter 5

A.1 Multiplexer Experiments

A.1.1 ACTT ratio results

"Dense" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999683 (0.003835)	0.999924 (0.003097)	0.999985 (0.002906)	0.999991 (0.002824)	0.999990 (0.002882)
BL2	0.999682 (0.003783)	0.999913 (0.003068)	0.999974 (0.002879)	0.999978 (0.002782)	0.999975 (0.002829)
"Dense" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	0.999973 (0.000378)	0.999996 (0.000315)	1.000000 (0.000299)	1.000000 (0.000283)	1.000000 (0.000278)
BL2	0.999966 (0.000378)	0.999984 (0.000307)	0.999983 (0.000268)	0.999982 (0.000230)	0.999984 (0.000209)
"Sparse" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999859 (0.002452)	0.999713 (0.003483)	0.999740 (0.003665)	0.999842 (0.003466)	0.999911 (0.003218)
BL2	0.999858 (0.002452)	0.999706 (0.003490)	0.999728 (0.003686)	0.999831 (0.003511)	0.999895 (0.003238)
"Sparse" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	0.999984 (0.000242)	0.999970 (0.000360)	0.999975 (0.000364)	0.999984 (0.000323)	0.999990 (0.000313)
BL2	0.999983 (0.000241)	0.999965 (0.000367)	0.999965 (0.000375)	0.999972 (0.000333)	0.999977 (0.000321)

Table A.1: Mean ACTT ratio (with standard deviation in brackets) for all bursts produced in the multiplexer experiments

A.2 Queue Experiments

A.2.1 Experimental runtimes

"Dense" burst production and burst cell size 10-1000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	77.36	69.65	1.77	1.73	1.90	1.82
5	237.31	213.04	6.68	6.35	6.53	6.12
10	555.22	481.16	37.17	211.61	36.05	214.44
15	745.23	640.99	83.59	611.49	81.06	627.88
20	945.48	794.09	151.61	1372.00	147.75	1404.03
"Dense" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	771.42	691.98	1.90	1.84	1.77	1.73
5	2377.09	2132.85	6.80	6.42	6.67	6.36
10	5687.06	4894.57	20.46	35.19	19.67	34.84
15	7411.87	6407.68	42.33	78.67	41.33	78.52
20	9344.42	7934.29	72.60	137.45	72.26	136.41
"Sparse" burst production and burst cell size 10-1000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	72.90	65.82	1.28	1.26	1.40	1.33
5	194.32	178.75	4.38	4.10	4.64	4.30
10	425.64	392.72	11.05	10.53	11.26	10.59
15	693.74	638.74	19.50	18.54	19.20	18.00
20	996.57	916.45	29.16	27.89	27.95	26.39
"Sparse" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	725.37	647.73	1.39	1.34	1.28	1.26
5	1930.23	1750.63	4.66	4.33	4.36	4.08
10	4255.65	3839.16	11.49	10.81	11.02	10.50
15	6939.29	6271.30	19.97	18.78	19.46	18.48
20	10007.58	9028.58	29.84	28.21	29.12	27.84

Table A.2: The actual runtimes (in seconds) of each experiment performed in the queue simulation tests

A.2.2 Relative burst start time differences

"Dense" burst production and burst cell size range 10-1000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	5326.77 (3645.90)	5448.87 (3665.18)	5356.76 (3675.22)	5470.74 (3683.47)
5	8504.42 (6590.98)	8637.88 (6710.35)	9909.94 (7919.83)	10035.60 (7860.05)
10	5459.76 (8989.82)	5656.35 (8815.39)	42712.30 (29773.60)	43820.40 (29959.60)
15	2755.64 (13445.40)	3262.54 (12591.30)	117212.66 (57259.40)	120054.23 (56193.10)
20	1238.22 (18872.90)	1803.55 (17717.60)	79996.40 (48681.30)	82853.30 (49516.80)
"Dense" burst production and burst cell size range 100-10000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	5152.37 (3513.73)	5231.67 (3522.68)	5291.94 (3695.83)	5383.99 (3717.38)
5	4582.22 (3365.63)	4641.27 (3358.55)	9377.86 (7149.35)	9553.72 (7256.81)
10	2207.61 (12713.50)	2462.63 (9753.25)	103724.63 (80781.00)	105526.85 (80904.60)
15	566.47 (31702.70)	1353.56 (14668.60)	176565.86 (108799.75)	183105.15 (107059.74)
20	-140.65 (35864.00)	824.34 (21154.60)	251462.32 (156045.42)	260531.50 (156492.87)

Table A.3: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst `START` time (cf. the cell-level results) for the burst-level simulators

"Sparse" burst production and burst cell size range 10-1000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	3120.29	3168.14	3119.50	3168.86
	(2287.20)	(2306.11)	(2285.33)	(2305.85)
5	4174.62	4283.08	4181.04	4289.66
	(2956.64)	(3053.55)	(2964.29)	(3085.82)
10	5072.17	5203.65	5150.79	5280.17
	(3478.46)	(3597.22)	(3541.18)	(3654.39)
15	5471.29	5566.40	5762.74	5938.62
	(4157.63)	(4283.26)	(4451.06)	(4363.27)
20	5860.34	6039.70	6369.37	6558.41
	(4823.47)	(4893.90)	(5224.23)	(5321.71)
"Sparse" burst production and burst cell size range 100-10000				
Stream	BL2		BL	
	Lowest	Highest	Lowest	Highest
2	3081.76	3133.73	3092.26	3135.10
	(2244.53	(2262.40)	(2262.60)	(2263.28)
5	4133.53	4196.24	4187.84	4259.67
	(2875.35)	(2969.03)	(2946.09)	(3027.54)
10	4680.36	4773.80	5116.03	5186.88
	(3346.27)	(3214.09)	(3679.07)	(3536.92)
15	4581.89	4704.34	5731.69	5874.64
	(3690.33)	(3341.72)	(4181.59)	(4225.90)
20	4194.13	4345.38	6313.22	6488.93
	(3650.26)	(3414.73)	(5297.92)	(5083.24)

Table A.4: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst `START` time (cf. the cell-level results) for the burst-level simulators

A.2.3 ACTT ratio results

"Dense" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999586 (0.004807)	0.999889 (0.004112)	1.009100 (0.097485)	1.020500 (0.161302)	1.028300 (0.232044)
BL2	0.999571 (0.004741)	0.999846 (0.004092)	1.009140 (0.099680)	1.020610 (0.166956)	1.028020 (0.227811)
"Dense" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	1.000020 (0.000502)	1.000110 (0.000464)	1.006170 (0.062326)	1.010680 (0.092262)	1.010090 (0.100441)
BL2	1.00000 (0.000510)	1.000060 (0.000499)	1.006090 (0.062197)	1.010590 (0.092113)	1.010000 (0.099531)
"Sparse" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999869 (0.003084)	0.999708 (0.004151)	0.999709 (0.004747)	0.999822 (0.004748)	0.999920 (0.004642)
BL2	0.999866 (0.003084)	0.999686 (0.004163)	0.999675 (0.004736)	0.999791 (0.004801)	0.999875 (0.004679)
"Sparse" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	1.000000 (0.000268)	1.000020 (0.000409)	1.000050 (0.000465)	1.000080 (0.000457)	1.000100 (0.000443)
BL2	0.999996 (0.000269)	1.000000 (0.000437)	1.000030 (0.000493)	1.000050 (0.000501)	1.000070 (0.000504)

Table A.5: Mean ACTT ratio (with standard deviation in brackets) for all bursts produced in the queue experiments

A.3 Demultiplexer Experiments

A.3.1 Experimental runtimes

"Dense" burst production and burst cell size 10-1000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	71.31	58.85	2.07	2.01	2.19	2.12
5	186.29	154.05	9.30	8.81	9.28	8.78
10	391.02	317.71	26.79	26.41	24.54	24.04
15	606.84	486.45	51.19	50.41	43.34	42.47
20	830.99	659.15	82.89	81.94	67.22	66.07
"Dense" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	709.76	584.35	2.05	2.02	2.21	2.15
5	1850.56	1531.94	9.26	8.84	9.56	9.04
10	3870.08	3161.05	26.98	26.51	26.81	26.19
15	5996.10	4862.82	51.44	50.43	49.15	47.89
20	8205.15	6581.65	83.18	82.08	77.63	76.19
"Sparse" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	71.12	58.57	1.45	1.41	1.54	1.49
5	181.17	153.18	5.21	4.88	5.50	5.09
10	369.39	317.00	13.91	13.56	14.15	13.43
15	562.51	484.51	25.47	24.25	25.06	23.96
20	761.36	654.85	39.21	37.71	37.68	36.40
"Sparse" burst production and burst cell size 100-10000						
Streams	CL	CLt	BL	BLt	BL2	BL2t
2	709.37	579.17	1.45	1.40	1.54	1.49
2	709.37	579.17	1.45	1.40	1.54	1.49
5	1800.41	1502.81	5.49	5.12	5.20	4.86
10	3651.21	3080.38	14.41	13.76	13.92	13.33
15	5544.54	4682.94	25.87	24.65	25.49	24.24
20	7473.77	6303.81	39.59	38.13	39.22	37.63

Table A.6: The actual runtimes (in seconds) of each experiment performed in the demultiplexer simulation tests

A.3.2 Relative burst start time differences

"Dense" burst production and burst cell size range 10-1000				
Stream	BL		BL2	
	Lowest	Highest	Lowest	Highest
2	5214.00 (3450.18)	5277.83 (3449.74)	5194.82 (3415.22)	5243.53 (3408.18)
5	10136.20 (9413.14)	12346.20 (10266.50)	8383.93 (6422.85)	10460.70 (7180.82)
10	55017.80 (49070.80)	63033.50 (52833.50)	6498.18 (5070.30)	11382.60 (7525.87)
15	32406.60 (19274.40)	41171.41 (20197.77)	4484.89 (3034.33)	12715.60 (8037.40)
20	65721.90 (35845.80)	78805.90 (40958.30)	7261.03 (6256.65)	14242.70 (11883.30)
"Dense" burst production and burst cell size range 100-10000				
Stream	BL		BL2	
	Lowest	Highest	Lowest	Highest
2	5179.01 (3275.78)	5216.80 (3411.68)	5024.73 (3072.88)	5027.25 (3170.58)
5	8938.68 (6306.38)	10946.50 (7056.63)	4486.86 (2461.19)	6189.35 (3671.12)
10	42540.80 (34671.70)	47993.10 (35637.80)	3119.72 (1576.15)	6350.08 (4126.67)
15	189117.21 (94894.40)	197938.76 (94118.10)	2917.4 (1546.58)	6024.07 (3189.47)
20	274147.05 (141429.78)	287780.79 (142696.85)	5102.58 (2466.64)	6912.67 (5024.35)

Table A.7: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst `START` time (cf. the cell-level results) for the burst-level simulators

"Sparse" burst production and burst cell size range 10-1000				
Stream	BL		BL2	
	Lowest	Highest	Lowest	Highest
2	3083.41	3086.34	3082.95	3089.39
	(2197.70)	(2161.85)	(2199.89)	(2168.48)
5	4131.54	4341.29	4114.86	4335.54
	(2799.07)	(2957.06)	(2778.37)	(2946.70)
10	5137.61	5588.38	5071.45	5533.25
	(3157.40)	(3516.71)	(3094.50)	(3440.70)
15	5823.25	6940.33	5573.73	6686.30
	(3721.13)	(4492.62)	(3469.87)	(4249.30)
20	7003.11	8505.51	6479.06	7922.75
	(4853.46)	(5577.23)	(4340.56)	(5016.42)
"Sparse" burst production and burst cell size range 100-10000				
Stream	BL		BL2	
	Lowest	Highest	Lowest	Highest
2	3054.95	3094.35	3052.62	3089.32
	(2141.39)	(2179)	(2136.17)	(2168.77)
5	4144.14	4349.72	4079.51	4270.68
	(2850.43)	(2976.78)	(2766.03)	(2855.45)
10	5094.36	5589.92	4631.17	5105.33
	(3194.90)	(3538.29)	(2737.26)	(3091.11)
15	5729.53	6909.84	4599.43	5603.71
	(3666.42)	(4467.47)	(2637.88)	(3270.90)
20	6838.05	8467.82	4590.39	5885.41
	(4710.36)	(5622.19)	(2811.07)	(3515.30)

Table A.8: Lowest and highest cell stream mean (with standard deviation in brackets) differences in burst `START` time (cf. the cell-level results) for the burst-level simulators

A.3.3 ACTT ratio results

"Dense" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999691 (0.003844)	1.000369 (0.003070)	1.000471 (0.003119)	1.000338 (0.003245)	1.000246 (0.003141)
BL2	0.999667 (0.003838)	1.000333 (0.003101)	1.000437 (0.003053)	1.000297 (0.003151)	1.000205 (0.003028)
"Dense" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	0.999968 (0.000375)	1.000038 (0.000296)	1.000050 (0.000310)	1.000036 (0.000315)	1.000027 (0.000308)
BL2	0.999941 (0.000410)	1.000015 (0.000358)	1.000036 (0.000320)	1.000020 (0.000296)	1.000009 (0.000285)
"Sparse" burst production and 10-1000 cells per burst					
Streams	2	5	10	15	20
BL	0.999833 (0.002438)	0.999804 (0.003494)	1.000028 (0.003781)	1.000219 (0.003488)	1.000327 (0.003275)
BL2	0.999828 (0.002441)	0.999782 (0.003517)	1.000000 (0.003788)	1.000182 (0.003502)	1.000293 (0.003322)
"Sparse" burst production and 100-10000 cells per burst					
Streams	2	5	10	15	20
BL	0.999985 (0.000248)	0.999980 (0.000344)	1.000004 (0.000374)	1.000024 (0.000337)	1.000034 (0.000321)
BL2	0.999980 (0.000253)	0.999965 (0.000362)	0.999980 (0.000392)	0.999996 (0.000367)	1.000006 (0.000364)

Table A.9: Mean ACTT ratio (with standard deviation in brackets) for all bursts produced in the demultiplexer experiments

A.3.4 Overlong burst results

"Dense" burst production, burst size range 10-1000 cells				
	BL		BL2	
Streams	Mean	sd	Mean	sd
2	0.000000	0.000000	0.000004	0.000220
5	0.235669	0.215881	0.238171	0.217308
10	0.255758	0.232152	0.260781	0.234892
15	0.255360	0.237722	0.260941	0.238377
20	0.255761	0.241134	0.259804	0.238182

"Dense" burst production, burst size range 100-10000 cells				
	BL		BL2	
Streams	Mean	sd	Mean	sd
2	0.000000	0.000000	0.006423	0.039098
5	0.258125	0.214288	0.300110	0.244778
10	0.278437	0.229278	0.306425	0.252190
15	0.279752	0.234527	0.296458	0.253657
20	0.275359	0.237131	0.287730	0.258937

"Sparse" burst production, burst size range 10-1000 cells				
	BL		BL2	
Streams	Mean	sd	Mean	sd
2	0.000000	0.000000	0.000000	0.000005
5	0.058164	0.160551	0.058959	0.161896
10	0.160155	0.221564	0.161473	0.223338
15	0.206576	0.225578	0.208347	0.227455
20	0.232094	0.226538	0.233937	0.227782

"Sparse" burst production, burst size range 100-10000 cells				
	BL		BL2	
Streams	Mean	sd	Mean	sd
2	0.000000	0.000000	0.001019	0.013463
5	0.061613	0.164101	0.067593	0.174853
10	0.172259	0.226278	0.185626	0.242648
15	0.219847	0.226551	0.240863	0.247691
20	0.247615	0.225644	0.272344	0.247143

Table A.10: The mean and standard deviation of the delay of `FINISH` messages generated by the `Demultiplexer` object which produce "overlong" bursts at the `Receiver` objects. The delay is calculated as a fraction of the ACTT value for each overlong burst (ie. a fraction of a cell)

Appendix B

Example code and simulation reports for Chapter 6

B.1 Burst-level simulation of a network (as performed in Section 6.2)

This section shows the top-level simulation code necessary to instantiate and configure the burst-level simulation objects necessary to perform one of the communication network simulation experiments for Section 6.2.

```
#include <math.h>
#include "sim.h"
#include "rand.h"
#include "ev_class.h"
#include "ents.h"
#include "queue_new.h"
#include "mux_new.h"
#include "demux_new.h"
#include "switch_new.h"
#include "VBRsrc.h"
#include "CBRsrc.h"
#include "CBRsink.h"

#include <stdlib.h>

//*******************************************
//**** Integer-time burst-level simulation ****
//*******************************************

//Chapter 6

reporter rep(DISK);

int main(int argc, char **argv)
{
```

```
int i;

//The global event list
ev_list *sim_ev_list;

//The pointers to the objects which will be used
switch_new *sw[4];          //The switches

CBRmultisrc *cbrsrc[1];     //The CBR source

mux_new *mux[1];            //Feeds the CBR streams into the net

VBRsrc *rs[3];              //X-connect traffic gen

simple_sink *rec[3];        //X-connect receivers

CBRsink *cbrsnk[3];         //The CBR receivers

//Build the event list
sim_ev_list = new ev_list();

//Build the simulation objects
sw[0] = new switch_new("Sw0", sim_ev_list, 4, TRUE, TRUE);
sw[1] = new switch_new("Sw1", sim_ev_list, 4, TRUE, TRUE);
sw[2] = new switch_new("Sw2", sim_ev_list, 4, TRUE, TRUE);
sw[3] = new switch_new("Sw3", sim_ev_list, 4, TRUE, TRUE);

cbrsrc[0] = new CBRmultisrc("CBRsrc", sim_ev_list, 10, 1000);

mux[0] = new mux_new("MovieMux", sim_ev_list, 10, 0, 1000, 0);

rs[0] = new VBRsrc("RS0", sim_ev_list, 0);
rs[1] = new VBRsrc("RS1", sim_ev_list, 0);
rs[2] = new VBRsrc("RS2", sim_ev_list, 0);

rec[0] = new simple_sink("Rec0", sim_ev_list, 3);
rec[1] = new simple_sink("Rec1", sim_ev_list, 3);
rec[2] = new simple_sink("Rec2", sim_ev_list, 3);

cbrsnk[0] = new CBRsink("CBRsink0", sim_ev_list, 10, 62.208, 1000);
cbrsnk[1] = new CBRsink("CBRsink1", sim_ev_list, 10, 62.208, 1000);
cbrsnk[2] = new CBRsink("CBRsink2", sim_ev_list, 10, 62.208, 1000);

//Configure switch 0
sw[0]->SetOPQueue(0, 4000, 4000, 1000);
sw[0]->SetOPQueue(1, 4000, 4000, 1000);
sw[0]->SetOPQueue(2, 4000, 7337+4000, 1000);
sw[0]->SetOPQueue(3, 4000, 4000, 1000);
```

```
mux[0]->SetLink(sw[0]->GetInPort(0));
sw[0]->SetOutPort(0, cbrsrc[0]);

sw[0]->SetOutPort(1, rec[0]);

sw[0]->SetOutPort(2, sw[1]->GetInPort(0));

rs[0]->SetLink(sw[0]->GetInPort(3));

sw[0]->AddressSize(20);
sw[0]->SetAddress(4, 1);
sw[0]->SetAddress(5, 2);
sw[0]->SetAddress(6, 2);
sw[0]->SetAddress(10, 0);
sw[0]->SetAddress(11, 2);
sw[0]->SetAddress(12, 2);
sw[0]->SetAddress(13, 2);

//Configure switch 1
sw[1]->SetOPQueue(0, 4000, 7336+4000, 1000);
sw[1]->SetOPQueue(1, 4000, 4000, 1000);
sw[1]->SetOPQueue(2, 1000, 36683+1000, 1000);
sw[1]->SetOPQueue(3, 62210, 62210, 1000);

sw[1]->SetOutPort(0, sw[0]->GetInPort(2));

rs[1]->SetLink(sw[1]->GetInPort(1));

sw[1]->SetOutPort(2, sw[2]->GetInPort(0));

sw[1]->SetOutPort(3, cbrsnk[2]);
cbrsnk[2]->SetLink(sw[1]->GetInPort(3));

sw[1]->AddressSize(20);
sw[1]->SetAddress(4, 0);
sw[1]->SetAddress(5, 2);
sw[1]->SetAddress(6, 2);
sw[1]->SetAddress(10, 0);
sw[1]->SetAddress(11, 2);
sw[1]->SetAddress(12, 2);
sw[1]->SetAddress(13, 3);

//Configure switch 2
sw[2]->SetOPQueue(0, 1000, 36683+1000, 1000);
sw[2]->SetOPQueue(1, 4000, 7336+4000, 1000);
sw[2]->SetOPQueue(2, 62210, 62210, 1000);
sw[2]->SetOPQueue(3, 4000, 4000, 1000);

sw[2]->SetOutPort(0, sw[1]->GetInPort(2));
```

```
sw[2]->SetOutPort(1, sw[3]->GetInPort(0));

sw[2]->SetOutPort(2, cbrsnk[1]);
cbrsnk[1]->SetLink(sw[2]->GetInPort(2));

sw[2]->SetOutPort(3, rec[1]);

sw[2]->AddressSize(20);
sw[2]->SetAddress(4, 0);
sw[2]->SetAddress(5, 3);
sw[2]->SetAddress(6, 1);
sw[2]->SetAddress(10, 0);
sw[2]->SetAddress(11, 1);
sw[2]->SetAddress(12, 2);
sw[2]->SetAddress(13, 0);

//Configure switch 3
sw[3]->SetOPQueue(0, 4000, 7336+4000, 1000);
sw[3]->SetOPQueue(1, 4000, 4000, 1000);
sw[3]->SetOPQueue(2, 4000, 4000, 1000);
sw[3]->SetOPQueue(3, 62210, 62210, 1000);

sw[3]->SetOutPort(0, sw[2]->GetInPort(1));

rs[2]->SetLink(sw[3]->GetInPort(1));

sw[3]->SetOutPort(2, rec[2]);

sw[3]->SetOutPort(3, cbrsnk[0]);
cbrsnk[0]->SetLink(sw[3]->GetInPort(3));

sw[3]->AddressSize(20);
sw[3]->SetAddress(4, 0);
sw[3]->SetAddress(5, 0);
sw[3]->SetAddress(6, 2);
sw[3]->SetAddress(10, 0);
sw[3]->SetAddress(11, 3);
sw[3]->SetAddress(12, 0);
sw[3]->SetAddress(13, 0);

//Configure the VBR X-traffic sources
rs[0]->SetFrom(0);
rs[0]->SetDest(6);
rs[0]->SetDists(1000, 100000, 4000, 62210, 0.0, 2.0, 1000000);

rs[1]->SetFrom(1);
rs[1]->SetDest(5);
rs[1]->SetDists(1000, 100000, 4000, 62210, 0.0, 2.0, 2000000);
```

```
rs[2]->SetFrom(2);
rs[2]->SetDest(4);
rs[2]->SetDists(1000, 100000, 4000, 62210, 0.0, 2.0, 3000000);

rs[0]->SetBursts(100000, 1);
rs[0]->SetFinTime(20000000000000ULL);
rs[1]->SetBursts(100000, 1);
rs[1]->SetFinTime(20000000000000ULL);
rs[2]->SetBursts(100000, 1);
rs[2]->SetFinTime(20000000000000ULL);

//Schedule WAKEUP messages for each VBR source
rs[0]->Start(0);
rs[1]->Start(0);
rs[2]->Start(0);

//Configure the CBR source
cbrsrc[0]->SetSRCAddress(10);
cbrsrc[0]->SetCBRREQsize(5);

//Configure the CBR receivers
cbrsnk[0]->SetNetAddress(11);
cbrsnk[1]->SetNetAddress(12);
cbrsnk[2]->SetNetAddress(13);

cbrsnk[0]->SetSRCAddress(10);
cbrsnk[1]->SetSRCAddress(10);
cbrsnk[2]->SetSRCAddress(10);

cbrsnk[0]->SetCBR_pkt_size(3500, 62.208);
cbrsnk[1]->SetCBR_pkt_size(3500, 62.208);
cbrsnk[2]->SetCBR_pkt_size(3500, 62.208);

cbrsnk[0]->SetCBRREQsize(5);
cbrsnk[1]->SetCBRREQsize(5);
cbrsnk[2]->SetCBRREQsize(5);

//Configure each channel in the CBR source
int *movie_channels = new int[10];
for(i = 0; i < 10; ++i)
{
  cbrsrc[0]->SetOutputLink(i, mux[0]);
  cbrsrc[0]->SetParams(i, i, 62.208, 3500, 10, 1000000 * (i+1), 0);
  movie_channels[i] = 10;
}
cbrsnk[0]->SetPktsPerCh(movie_channels);
cbrsnk[1]->SetPktsPerCh(movie_channels);
cbrsnk[2]->SetPktsPerCh(movie_channels);
```

```
delete [] movie_channels;

//Set a random wait distribution in each CBR receiver
cbrsnk[0]->SetWaitDistro(1.0);
cbrsnk[1]->SetWaitDistro(1.0);
cbrsnk[2]->SetWaitDistro(1.0);

//Set a finish time for each CBR receiver
cbrsnk[0]->SetFinTime(2000000000000ULL);
cbrsnk[1]->SetFinTime(2000000000000ULL);
cbrsnk[2]->SetFinTime(2000000000000ULL);

//Schedule a WAKEUP message for time 0 to start each CBR receiver
cbrsnk[0]->Start(0);
cbrsnk[1]->Start(0);
cbrsnk[2]->Start(0);

//Set an intermittent ''alarm clock'' to monitor simulation progress
sim_ev_list->SetAlarm(0, 500000000000ULL, 2000000000000ULL);

//Run the simulation (terminate at max possible sim time)
sim_ev_list->run_sim(18446744073709551610ULL);

//Delete the global event list - will report unprocessed events (if nec.)
delete sim_ev_list;

//Delete the simulation objects which triggers the generation of reports.
delete sw[0];
delete sw[1];
delete sw[2];
delete sw[3];

delete mux[0];

delete rs[0];
delete rs[1];
delete rs[2];

delete rec[0];
delete rec[1];
delete rec[2];

delete cbrsrc[0];

delete cbrsnk[0];
delete cbrsnk[1];
delete cbrsnk[2];
```

```
  return 0;
}
```

B.2 Example simulation report file (for the high speed network model in Section 6.4)

This section shows a simulation report file from one of the experiments performed for the SynchroLan model in Section 6.4. The workstation bus simulated in this example was the PCI bus.

```
                  /-----------------------------------\
                  | Integer-time burst-level simulation |
                  \-----------------------------------/
Simulation terminated at time = 18446744073709551610.
Total number of events processed           : 4791099.
Max events pending                         : 14.
Total number of entities in the simulation : 20.

Performance:
  simulation started  : Wed Mar 1 21:40:32 2000
  simulation finished : Wed Mar 1 21:41:12 2000
  total run time      : 40 seconds.
  Events/sec          : 119777

There were 250283 event indices and av ev per index = 19.1427

The following reports are from individual entities:


PC0 received a total of 100505 packets and sent 100000.
Terminated at 8.1544 (seconds)
Total time = 8.1544
Think time = 20.4512%
Idle time  = 0%
Send time  = 37.9983%
Rec time   = 41.5505%

InQueue0 received 411668480 bytes, sent 411668480 bytes and lost 0 bytes.
CLR = 0
Received: 110611 bursts (0 were zero bursts) and produced: 115641 bursts.

InMux0 sent a total of 411668480 bytes and received:
  0 bytes on input port 0.
  137805824 bytes on input port 1.
  136671232 bytes on input port 2.
  137191424 bytes on input port 3.
```

```
Tot bytes rec = 411668480
Received: 100505 bursts (0 were zero bursts) and produced: 110611 bursts.

OutQueue0 received and sent 99997 packets.

OutDemux0 received a total of 409587712 bytes and sent:
  0 bytes in 0 bursts on output port 0.
  136716288 bytes in 33378 bursts on output port 1.
  136146944 bytes in 33239 bursts on output port 2.
  136724480 bytes in 33380 bursts on output port 3.
Received: 99997 bursts (0 were zero bursts) and produced: 99997 bursts
 (including 0 cancelled zero output bursts).
Comm channel utilisation = 16.0497%

PC1 received a total of 100138 packets and sent 100399.
Total time = 8.15441
Think time = 20.44%
Idle time  = 0%
Send time  = 38.1511%
Rec time   = 41.4089%

InQueue1 received 410165248 bytes, sent 410165248 bytes and lost 0 bytes.
CLR = 0
Received: 110212 bursts (0 were zero bursts) and produced: 115132 bursts.

InMux1 sent a total of 410165248 bytes and received:
  136716288 bytes on input port 0.
  0 bytes on input port 1.
  137404416 bytes on input port 2.
  136044544 bytes on input port 3.
Tot bytes rec = 410165248
Received: 100138 bursts (0 were zero bursts) and produced: 110212 bursts.

OutQueue1 received and sent 100399 packets.

OutDemux1 received a total of 411234304 bytes and sent:
  137805824 bytes in 33644 bursts on output port 0.
  0 bytes in 0 bursts on output port 1.
  136507392 bytes in 33327 bursts on output port 2.
  136921088 bytes in 33428 bursts on output port 3.
Received: 100399 bursts (0 were zero bursts) and produced: 100399 bursts
 (including 0 cancelled zero output bursts).
Comm channel utilisation = 16.1141%

PC2 received a total of 100037 packets and sent 100327.
Total time = 8.15441
Think time = 20.4738%
Idle time  = 0%
Send time  = 38.1237%
```

```
Rec time    = 41.4025%

InQueue2 received 409751552 bytes, sent 409751552 bytes and lost 0 bytes.
CLR = 0
Received: 109939 bursts (0 were zero bursts) and produced: 114751 bursts.

InMux2 sent a total of 409751552 bytes and received:
  136146944 bytes on input port 0.
  136507392 bytes on input port 1.
  0 bytes on input port 2.
  137097216 bytes on input port 3.
Tot bytes rec = 409751552
Received: 100037 bursts (0 were zero bursts) and produced: 109939 bursts.

OutQueue2 received and sent 100327 packets.

OutDemux2 received a total of 410939392 bytes and sent:
  136671232 bytes in 33367 bursts on output port 0.
  137404416 bytes in 33546 bursts on output port 1.
  0 bytes in 0 bursts on output port 2.
  136863744 bytes in 33414 bursts on output port 3.
Received: 100327 bursts (0 were zero bursts) and produced: 100327 bursts
 (including 0 cancelled zero output bursts).
Comm channel utilisation = 16.1026%

PC3 received a total of 100222 packets and sent 100179.
Total time = 8.15444
Think time = 20.4208%
Idle time  = 0%
Send time  = 38.0673%
Rec time   = 41.512%

InQueue3 received 410509312 bytes, sent 410509312 bytes and lost 0 bytes.
CLR = 0
Received: 110464 bursts (0 were zero bursts) and produced: 115482 bursts.

InMux3 sent a total of 410509312 bytes and received:
  136724480 bytes on input port 0.
  136921088 bytes on input port 1.
  136863744 bytes on input port 2.
  0 bytes on input port 3.
Tot bytes rec = 410509312
Received: 100222 bursts (0 were zero bursts) and produced: 110464 bursts.

OutQueue3 received and sent 100179 packets.

OutDemux3 received a total of 410333184 bytes and sent:
  137191424 bytes in 33494 bursts on output port 0.
  136044544 bytes in 33214 bursts on output port 1.
```

```
  137097216 bytes in 33471 bursts on output port 2.
  0 bytes in 0 bursts on output port 3.
Received: 100179 bursts (0 were zero bursts) and produced: 100179 bursts
 (including 0 cancelled zero output bursts).
Comm channel utilisation = 16.0793%

End of report file.
```

Bibliography

[1] M. Ajmone Marsan, A. Bianco, T.V. Do, L. Jereb, R. Lo Cigno, and M. Munafò. ATM Simulation with CLASS. *Performance Evaluation*, 24:137–159, 1995.

[2] M. Ajmone Marsan, C. Casetti, M. Munafò, and J. Valdes. Fair queueing in ATM networks: A simulation study. In *3rd IFIP Workshop on Performance Modelling and Evaluation of ATM Networks*, 1995.

[3] I.F. Akyildiz, I. Joe, R.M. Fujimoto, and I. Nikolaidis. Parallel simulation of end-to-end ATM models. *Computer Networks and ISDN Systems*, 29(6):873–877, 1997.

[4] *AMD-K6-2 Processor Code Optimization Application Note.* Advanced Micro Devices Inc., `http://www.amd.com`, February 2000.

[5] *AMD-K6-2 Processor Data Sheet.* Advanced Micro Devices Inc., `http://www.amd.com`, January 2000.

[6] C.J. Ani. Simulation technique for evaluating cell loss rate in an ATM network. *Simulation*, 64(5):320–329, 1995.

[7] R.L. Bagrodia. Perils and pitfalls of parallel discrete-event simulation. In *Proceedings of the conference on Winter simulation*, page 136, 1996.

[8] J. Banks, J.S. Carson, and B.L. Nelson. *Discrete Event System Simulation.* Prentice-Hall, New Jersey, 2nd edition, 1996.

[9] B. Bashforth and C. Williamson. Statistical Multiplexing of Self-Similar Video Streams: Simulation Study and Performance Results. In *Proc. of the 6th International Symposium on the Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS'98)*, pages 119–126, Montreal, July 1998.

[10] S. Bhatt, R. Fujimoto, A. Ogielski, and K. Perumalla. Parallel simulation techniques for large-scale networks. *IEEE Communications Magazine*, 36(8):42–47, August 1998.

[11] G.M. Birtwistle. *DEMOS: A System for Discrete Event Modelling on SIMULA.* Macmillan, 1979.

[12] G.M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA BEGIN.* Studentletteratur, Lund, Sweden, 1973.

[13] K.R. Bisset. An adaptive synchronization protocol for parallel discrete event simulation. In *31st Annual Simulation Symposium*, pages 26–33, April 1998.

[14] B. Blightman. Why are we doing this? (Simulation). In *IEE Colloquium on Discrete Event Simulation*, pages 1–2, 1991.

[15] M. Bocci, J.M. Pitts, and E.M. Scharf. Performance of time stepping mechanism for parallel cell rate simulation of ATM networks. In *IEE 11th Teleptraffic Symposium*, page 15B/15B/9, 1994.

[16] P. Bratley, B. Fox, and L. Schrage. *A Guide to Simulation (2nd Edition).* Springer-Verlag, London, 1987.

[17] G. Brebner. *Computers in Communication.* McGraw-Hill, 1997.

[18] G. Brebner and R. Pooley. ECOLE: A Configurable Environment for a Local Optical Network of Workstations. In *(CANPC'98) Springer LNCS*, number 1362, pages 44–58, February 1998.

[19] W. Brissinck and E. Dirkx. Analysis of large ATM switches using a platform-independent simulation environment. *Journal of Systems Architecture*, 44(6):411–431, 1998.

[20] C. Casetti, G. Favalessa, M. Mellia, and M.M. Munafò. An Adaptive Routing Algorithm for Best-effort Traffic in Integrated-Services Networks. In *16th International Teletraffic Congress (ITC-16),*, Edinburgh, UK, June 1999.

[21] A. Catsoulis, Y. Kamaras, K. Kavidopoulos, and N. Mitrou. An adaptive shaper with effective-rate enforcement for ATM traffic. In *Computers and Communications ISCC'98*, pages 19–23, 1998.

[22] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1979.

[23] J.G. Cleary and Jya-Jang Tsai. Conservative parallel simulation of ATM networks. In *Proceedings. Tenth Workshop on Parallel and Distributed Simulation. IEEE PADS 96*, pages 30–8, 1996.

[24] P.S. Coe, F.W. Howell, R.N. Ibbett, and L.M. Williams. A Hierarchical Computer Architecture Design and Simulation Environment. *ACM Transactions on Modeling and Computer Simulation*, 8(4):431–446, 1998.

[25] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[26] Jade Simulations International Corp. *Sim++ User Manual.* JSI Corp., 1992.

[27] D. Cotter, J.K. Lucek, and D.D. Marcenac. Ultra-High-Bit-Rate networking: From the Transcontinental Backbone to the Desktop. *IEEE Communications*, April 1997.

[28] S.D. Cusack. Simulating the Edinburgh Sparse Vector Processor using HASE. M.Sc. dissertation, University of Edinburgh, Department of Computer Science, September 1994.

[29] S.D. Cusack and R.J. Pooley. Developing an Object-Oriented Simulation Framework for ATM Network Performance Studies. In M. Merabti, M. Carew, and F. Ball, editors, *Proc. of UKPEW'95*, pages 64–80. Springer-Verlag, September 1995.

[30] S.D. Cusack and R.J. Pooley. Integer Time Based Burst-Level Simulation of Communication Networks. In P. Luker, editor, *Proc. of UKSim '97*, pages 159–164. UK Simulation Society, April 1997.

[31] S.D. Cusack and R.J. Pooley. Ultrafast simulation of high bandwidth networks. In *ATM '98*, Ilkley, July 1998. IFIP.

[32] S.R. Das. Estimating the cost of throttled execution in time warp. In *Proceedings. Tenth Workshop on Parallel and Distributed Simulation. IEEE PADS 96*, pages 196–9, 1996.

[33] M. De Prycker. *Asynchronous Transfer Mode.* Addison-Wesley, 1996.

[34] M. Devetsikiotis and J.K. Townsend. On the efficient simulation of large communication networks using importance sampling. In *GLOBECOM '92. Communication for Global Users. IEEE Global Telecommunications Conference.*, volume 3, pages 1455–9, 1992.

[35] A.K. Erlang. Probability and telephone calls. *Nyt Tidsskr. Mat.*, Ser. B(20):33–39, 1909.

[36] H.I. Fahmy and C. Douligeris. NAMS: network automated modeler and simulator. In *Proc. of 29th Annual Simulation Symposium*, pages 65–70, 1996.

[37] G.S. Fishman. *Concepts and methods in discrete event digital simulation.* Wiley, 1973.

[38] P.A. Fishwisk. Computer simulation: growth through extension. In *European Simulation Multiconference*, Barcelona, Spain, 1994.

[39] *User-Network Interface (UNI) Specification.* ATM Forum, `http://www.atmforum.com`, 1997.

[40] W.R. Franta. *The Process View of Simulation.* Elsevier, New York, 1977.

[41] W.R. Franta and K. Maly. An efficient data structure for the simulation event set. *Communications of the A.C.M.*, 20(8):596–602, 1977.

[42] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33:30–53, 1990.

[43] J. Garcia, Marin-Sillué, and J.L. Melús-Moreno. Description of a simulation environment to evaluate high performance ATM fast packet switches. *IFIP Transactions C - Communication Systems*, 26:423–438, 1994.

[44] P. Gburzynski, T. Ono-Tesfaye, and S. Ramaswamy. Modeling ATM networks in a parallel simulation environment: a case study. In *Proceedings of the 1995 Summer Computer Simulation Conference.*, pages 869–74, 1995.

[45] A. Gersht, G. Pathak, and A. Shulman. Burst level congestion control in ATM networks. In *3rd IEEE Symposium on Computers and Communications*, pages 258–264, 1998.

[46] T.J. Gibson and E.L. Miller. The Case for Personal Computers as Workstations. In *22nd International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems*, volume 2, pages 644–652, San Diago, 1996. CMG.

[47] F. Gomes, S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. SimKit: A High Performance Logical Process Simulation Class Library in C++. In *Proc. of the 1995 Winter Simulation Conference (WSC'95)*, pages 706–713, Arlington, Virginia, December 1995.

[48] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126, June 1982.

[49] C. Greenhalgh, S. Benford, A. Bullock, N. Kuijpers, and K. Donkers. Predicting network traffic for collaborative virtual environments. *Computer Networks and ISDN Systems*, 30(24):2293–2308, 1998.

[50] P. Gunning, J.K. Lucek, D.G. Moodie, K. Smith, D. Pitcher, Q. Badat, and A.S. Siddiqui. 40 Gbit/s optical-TDMA LAN over 300 metres installed blown fibre. In *ECOC '97*. IEE, September 1997.

[51] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), October 1996.

[52] T.R. Halfhill. Inside IA-64. *Byte*, 23(6):81–84, June 1998.

[53] J. Hall, R. Sabatino, S. Crosby, I. Leslie, and R. Black. A comparative study of high speed networks. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications.*, volume 2, pages 774–82, 1998.

[54] R. Händel, M.N. Huber, and S. Schröder. *ATM Networks - Concepts, Protocols, Applications.* Addison-Wesley, 2nd edition, 1994.

[55] F. Hao, I. Nikolaidis, and E.W. Zegura. Efficient simulation of ATM networks with accurate end-to-end delay statistics. In *IEEE International Conference on Communications*, volume 3, pages 1799–1804, 1998.

[56] G. Hipper and D. Tavangarian. A concurrent network architecture for cost-efficient parallel computing using workstation clusters. *Simulation Practice and Theory*, 6(2):197–218, 1998.

[57] F. Howell and R. McNab. simjava: a discrete event simulation package for java with applications in computer systems modelling. In *Proceedings First International Conference on Web-based Modelling and Simulation*, San Diego, January 1998. Society for Computer Simulation.

[58] Y. Huang and R.K. Iyer. An object-oriented environment for fast simulation using compiler techniques. In *WSC'98 - Winter Simulation Conference*, Washington, DC, 1998.

[59] R.N. Ibbett and N.P. Topham. *Architecture of High Performance Computers*, volume 1. Macmillan, 1989.

[60] *Pentium II Processor Developer's Manual.* Intel Corporation, `http://www.intel.com`, 1997.

[61] *Accelerated Graphics Port Interface Specification Revision 2.0.* Intel Corporation, `http://developer.intel.com/technology/agp/`, May 1998.

[62] *Intel Architecture Optimization.* Intel Corporation, `http://www.intel.com`, 1999.

[63] *B-ISDN User-Network Interface - Recommendation : I.413.* ITU-T, 1993.

[64] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[65] J.A. Joines and S.D. Roberts. Simulation in an object-oriented world. In *WSC'99 - Winter Simulation Conference*, 1999.

[66] G. Karlsson. Asynchronous transfer of video. Technical Report R95-14, Swedish Institute of Computer Science, 1995.

[67] P. Keleher. *Lazy Release Consistency for Distributed Shared Memeory.* PhD thesis, Rice University, Houston, Texas, 1995.

[68] P.J.B King. *Computer and Communication Systems Performance Modelling.* Prentice Hall, 1990.

[69] D.E. Knuth. *The Art of Computer Programming - volume 3 / Sorting and Searching.* Addison-Wesley, 1973.

[70] K. Kosbar and K. Schneider. Object-oriented modelling of communication systems. *IEE Colloquium on Computer modelling of communications systems*, 1994(15):1/1–8, 1994.

[71] W. Kreutzer and K. Osterbye. BetaSIM a framework for discrete event modelling and simulation. *Simulation Practice and Theory*, 6(6):573–599, 1998.

[72] C.Y. Kuen, R. Lo, and S. Husein. Performance evaluation of video signals at burst level in an ATM environment. pages 493–497, 1995.

[73] Y-B. Lin and P.A. Fishwick. Asynchronous parallel discrete event simulation. *IEEE Transactions on systems, man, and cyberspace-Part A: Systems and Humans*, 26(4):397–412, July 1996.

[74] Y-B. Lin and E.D. Lazowska. A study of time warp rollback mechanisms. *ACM Transactions on Modeling and Computer Simulation*, 1(1):51–72, 1991.

[75] M.C. Little and D.L. McCue. Construction and use of a simulation package in C++. *C User's Journal*, 12(3), March 1994.

[76] R. Lo Cigno and M. Munafò. RC - A flexible Language for the Specification of ATM Networks Simulation Experiments. In D. Kouvatosos, editor, *3rd IFIP workshop on the performance modelling and evaluation of ATM networks*, Ilkley, UK, July 1995.

[77] M.H. MacDougall. *Simulating Computer Systems - Techniques and Tools.* The MIT Press, 1987.

[78] M. Malgosa-Sanahuja, J. Castells-Cuscullola, and J. Garcia-Haro. An ATM switch simulation tool based on the C++ object-oriented programming language. In *IEEE Pacific Rim Conference on Comms, Computers and Signal Processing*, volume 2, pages 972–976, August 1997.

[79] M.A. Marsan, A. Bianco, C. Casetti, P. Castelli, R. Lo Cigno, M. Mellia, and M.M. Munafò. A CAC algorithm supporting different QoS classes. In D. Kouvatsos, editor, *Performance Modelling and Evaluation og ATM Networks*, volume 4. Kluwer Academic, 1999.

[80] M.A. Marsan, A. Bianco, C. Casetti, C-F Chiasserini, A. Francini, R. Lo Cigno, M. Mellia, and M. Munafò. An integrated software environment for the simulation of ATM networks. In *1997 Summer Computer Simulation Conference*, Arlington, Virginia, USA, July 1997.

[81] S.C. Mathewson. Simulation program generators. *Simulation*, 23(6):181–189, 1974.

[82] A.R. Mikler, J.S.K. Wong, and V. Honavar. An object-oriented approach to simulating large communication networks. *Journal of Systems and Software*, 40(2):151–164, February 1998.

[83] I. Mitrani. *Simulation techniques for discrete event systems.* Cambridge University Press, 1982.

[84] M. Naldi and F. Calonico. A comparison of the GEVT and RESTART techniques for the simulation of rare events in ATM networks. *Simulation Practice and Theory*, 6(2):181–196, 1998.

[85] K.M. Nichols. Improving network simulation with feedback. In *Proceedings 23rd Annual IEEE Conference on Local Computer Networks*, pages 208–21, 1998.

[86] I. Nikolaidis, R. Fujimoto, and C.A. Cooper. Parallel simulation of high-speed network multiplexers. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, volume 3, pages 2224–2229, 1993.

[87] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24, August 1991.

[88] C. Partridge. *Gigabit Networking.* Addison-Wesley, 1994.

[89] R. Pasquini and V. Rego. Optimistic parallel simulation over a network of workstations. In *WSC'99 - Winter Simulation Conference*, 1999.

[90] J.M. Pitts. *Cell-rate simulation modelling of asynchronous transfer mode telecommunication networks.* PhD thesis, Queen Mary & Westfield College, London, 1993.

[91] J.M. Pitts. Cell-rate modelling for accelerated simulation of ATM at the burst level. *IEE Communications*, 142(6):379–385, December 1995.

[92] R.J. Pooley. Object-oriented simulation models for performance modelling. In *11th UK Performance Engineering Workshop.* Springer-Verlag, September 1995.

[93] R.J. Pooley and G.M Birtwistle. A Process Based Simulation of X.25 using DEMOS. In *Proceedings of SCS Conference on Simulation and Strongly Typed Languages*, pages 127–131, San Diego, January 1984.

[94] D.P. Sanderson and L.L. Rose. Object-oriented modeling using C++. In *Proceedings of the 21st annual conference on Simulation symposium*, pages 143–156, Tampa, FL USA, March 1988. ACM.

[95] T.J. Schriber and D.T. Brunner. Inside discrete-event simulation software: How it works and why it matters. In *Winter Simulation conference*, volume 1, pages 77–85, 1998.

[96] E. Shapiro. Overview: New directions for simulation and modeling of complex systems. *Simulation Practice and Theory*, 6(2):91–97, 1998.

[97] A. Srivastava, A. Kumar, A. Singru, and K.A. Kamel. Object-oriented simulation of multimedia on demand services over ATM network. In *Proceedings of IEEE MASCOTS '96*, pages 128–132, 1996.

[98] J.S. Steinman. Discrete-event simulation and the event horizon. In *8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 39–49, 1994.

[99] J.S. Steinman. Discrete-event simulation and the event horizon. 2. event list management. In *Proceedings. Tenth Workshop on Parallel and Distributed Simulation. PADS 96*, pages 170–8, 1996.

[100] D. Stiliadis and A. Varma. FAST: an FPGA-based simulation testbed for ATM networks. In *IEEE International Conference on Communications*, volume 1, pages 374–378, June 1996.

[101] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[102] E.L. Taylor Jr. and S.F. Midkiff. Simulation and analysis of an ATM video-on-demand network using available bit rate service. In *Proceedings 7th IEEE International Conference on Computer Communications and Networks*, pages 462–9., 1998.

[103] Y-M. Teo and S-C. Tay. Performance evaluation of a parallel simulation environment. In *32nd Annual Simulation Symposium*, pages 86–93, April 1999.

[104] J.K. Townsend, Z. Haraszti, J.A. Freebersyser, and M. Devetsikiotis. Simulation of rare events in communications networks. *IEEE Communications Magazine*, 36(8):36–41, August 1998.

[105] P. Tsingotjidis and J.F. Hayes. Proactive congestion controls for real time services. In *Electrical and Computer Engineering 1997*, volume 2, pages 808–811, Canada, 1997.

[106] R. Ulrich, U. Herzog, and P. Kritzinger. Modeling buffer utilization in cell-based networks. *Performance Evaluation*, 31(3), 1998.

[107] B. Unger. ATM-TN system design. Technical report, WurcNet Inc., 1994.

[108] *YATS - Yet Another Tiny Simulator.* University of Dresden, `http://www.ifn.et.tu-dresden.de/TK/yats`.

[109] J.G. Vaucher and P. Duval. A comparison of simulation event list algorithms. *Communications of the A.C.M.*, 18(4):223–230, 1975.

[110] M. Villén-Altamirano, A. Matinez-Marròn, J. Game, and F. Fernandez-Cuesta. Enhancement of the accelerated simulation method RESTART by considering multiple thresholds. In *Proc. of the 14th International Teletraffic Congress*, volume 1a, pages 797–810, Antibes Juan-les-Pins, France, 1994.

[111] M. Villén-Altamirano and J. Villén-Altamirano. Restart: A method for accelerating rare event simulations. In *Proc. of the 13th International Teletraffic Congress*, pages 71–76, Copenhagen, Denmark, 1991.

[112] L. Wang and C. McCrosky. Performance comparison of control schemes for ABR service in ATM LANs. In *MASCOTS '97*, pages 205–212, 1997.

[113] P.E. Wirth. The rôle of teletraffic modeling in the new communications paradigms. *IEEE Communications Magazine*, 35(8):86–92, 1997.

[114] F.P. Wyman. Improved event-scanning mechanisms for discrete event simulation. *Communications of the A.C.M.*, 18(6):350–353, 1975.

[115] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation. In *Proc. of the 13th Workshop on Parallel and Distributed Simulation (PADS99)*, Atlanta, Georgia, May 1999.

[116] Z. Yao and D.C. Blight. Modeling and simulation of ATM networks. In *IEEE WESCANEX'97*, pages 202–207, May 1997.